

原书第3版

计 算 机 科 学 丛 书

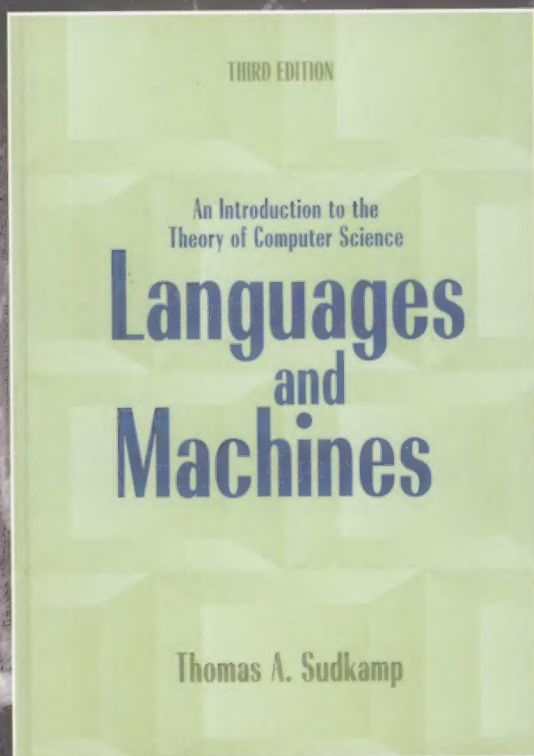
HZ BOOKS
华章教育

PEARSON
Addison
Wesley

语言与机器

计算机科学理论导论

(美) Thomas A. Sudkamp 著 孙家骥 等译



Languages and Machines
An Introduction to the Theory of Computer Science
Third Edition



机械工业出版社
China Machine Press

语言与机器 计算机科学理论导论 (原书第3版)

理论计算机科学是推动计算机技术和应用向前发展的巨大动力。形式语言、自动机、可计算性、计算复杂性和相关方面内容构成的计算理论,是理论计算机科学的基础内容之一。本书由美国莱特州立大学计算机科学及工程系的Thomas A. Sudkamp教授编写,是介绍这些内容的优秀教材。

全书不仅介绍了计算机科学的基础,探讨了算法计算的能力和局限;而且还通过概念的严格表述,以及使用通俗的例子来解释定理,从而帮助学生提高数学论证能力。书中每章后面都有一些练习,通过这些练习使学生加深对本章内容的理解。

第三版亮点:

- 超过100个新的习题和示例
- 强调问题表述和问题归约的重要性
- 显著地扩展了计算复杂性的覆盖面
- 增加了用问题归约证明NP完全性的章节
- 处理NP完全问题时给出了近似算法

作者简介

Thomas A. Sudkamp

是美国莱特州立大学计算机科学及工程系的教授,他的研究领域广泛,包括近似推理、人工智能、数理逻辑、建模软计算的应用、复杂问题领域的决策制定以及不确定、不精确信息和知识发掘的机器学习。Sudkamp教授目前还担任IEEE Transactions on System, Man, and Cybernetics和IEEE Transactions on Fuzzy Systems的副编辑, International Journal of Approximate Reasoning和Fuzzy Sets and Systems的领域编辑。他也曾经担任过北美模糊信息处理协会 (NAFIPS) 的主席以及国际模糊系统联盟 (IFSA) 的副主席。



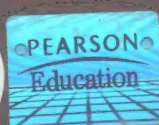
www.PearsonEd.com

投稿热线: (010) 88379604
购书热线: (010) 68995259, 68995264
读者信箱: hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计: 包旭 林杉



上架指导: 计算机/计算机理论

ISBN 978-7-111-22634-5



9 787111 226345

ISBN 978-7-111-22634-5
定价: 49.00 元

计 算 机

TP3/560

2008

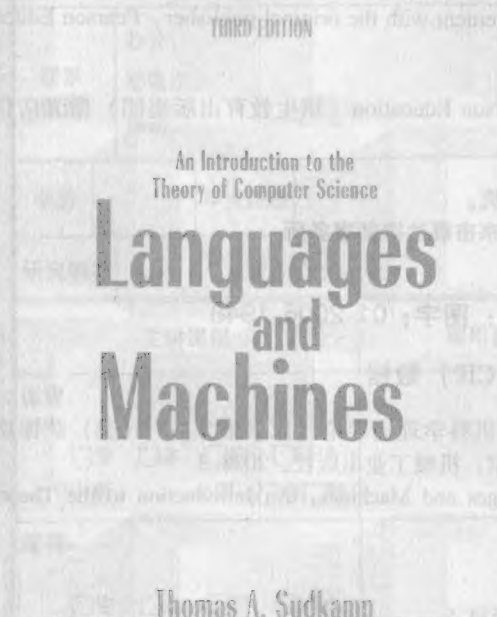
书

原书第3版

语言与机器

计算机科学理论导论

(美) Thomas A. Sudkamp 著 孙家骥 等译



Languages and Machines
An Introduction to the Theory
of Computer Science
Third Edition



机械工业出版社
China Machine Press

本书是计算理论方面的优秀教材之一,包括上下文无关文法、上下文无关文法范式、有限自动机、正则语言的性质、下推自动机和上下文无关语言、图灵机、图灵可计算函数、乔姆斯基层次、判定问题与丘奇图灵机、不可判定性、 μ -递归函数、时间复杂性、库克定理、NP-完全问题、LL(k)文法以及LR(k)文法等问题。本书不仅介绍了计算机科学的基础,而且通过概念的严格表述,以及使用通俗的例子来阐释定理,从而帮助学生提高数学论证能力以及对计算理论知识的全面深入的理解。书中每章后面都有附有大量习题,通过完成这些习题,学生可以加深对本章内容的理解。

本书可以用作计算机科学、计算机工程及其相关专业的教材,也可以作为从事计算理论、形式语言以及计算机系统研发的研究人员和工程技术人员的参考书。

Simplified Chinese edition copyright © 2008 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Languages and Machines: An Introduction to the Theory of Computer Science, Third Edition* (ISBN 0-321-32221-5) by Thomas A. Sudkamp, Copyright © 2006 by Pearson Education, Inc.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison Wesley.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签,无标签者不得销售。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2006-1948

图书在版编目(CIP)数据

语言与机器:计算机科学理论导论(原书第3版)/(美)萨德坎普(Sudkamp, T. A.)著;孙家骕等译. —北京:机械工业出版社,2008.3

书名原文: *Languages and Machines: An Introduction to the Theory of Computer Science, Third Edition*

(计算机科学丛书)

ISBN 978-7-111-22634-5

I. 语… II. ①萨… ②孙… III. 计算机科学 IV. TP3

中国版本图书馆CIP数据核字(2007)第164490号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:李南丰

北京诚信伟业印刷有限公司印刷·新华书店北京发行所发行

2008年3月第1版第1次印刷

184mm×260mm·25.75印张

标准书号:ISBN 978-7-111-22634-5

定价:49.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换
本社购书热线(010)68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅壁划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总体规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程,而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下,读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证,但我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

电子邮件: hzsj@hzbook.com

联系电话: (010) 68995264

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元	王 珊	冯博琴	史忠植	史美林
石教英	吕 建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范 明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅 宏	程 旭	程时端	谢希仁
裘宗燕	戴 葵			

食 物 毒 害

随着人们生活水平的提高，饮食结构发生了显著变化，食物安全问题日益突出。食物中毒是指摄入了含有生物性、化学性有毒物质的食物后，出现的急性或慢性中毒。食物中毒的常见类型包括细菌性食物中毒、化学性食物中毒、真菌性食物中毒等。预防食物中毒的关键在于加强食品卫生管理，确保食物来源可靠，加工过程规范，储存条件适宜。同时，公众也应提高食品安全意识，养成良好的饮食习惯，避免食用过期、变质食物。



译者序

理论计算机科学是推动计算机技术和应用向前发展的巨大动力。形式语言、自动机、可计算性、计算复杂性以及相关内容构成的计算理论,是理论计算机科学的基础内容之一。学习、研究这些内容,不仅为进一步学习、研究理论计算机科学打下基础,而且对增强形式化能力和推理能力也有帮助作用,这些能力对从事计算机技术中的形式化方法、技术等研究,是不可缺少的。

本书是由美国莱特州立大学计算机科学及工程系的 Thomas A. Sudkamp 教授编写的高等学校教材,主要介绍形式语言、自动机、可计算性、计算复杂性和上下文无关语言的确定性分析等内容。本书不仅介绍了计算机科学的基础,而且通过概念的严格表述,以及使用通俗的例子来阐释定理,从而期望能够帮助学生提高数学论证能力以及对计算理论知识的全面深入的理解。书中每章后面都有大量习题,通过完成这些习题,学生可以加深对本章内容的理解。本书是理论计算机科学的优秀教材之一。

本书可以用作计算机科学、计算机工程及其相关专业的教材,也可以作为从事计算理论、形式语言以及计算机系统研发的研究人员和工程技术人员的参考书。

引进国外的优秀计算机教材,对我国的计算机教育事业的发展会起到积极的推动作用,也是与世界接轨、建立世界一流大学不可缺少的条件之一。我们把本书介绍给国内从事计算机教育事业的同行们,以供参考。

参加本书翻译的有:孙家骥同志,负责各章译稿的详细修改和全书的统稿;郝丹同志负责第1到第4章及前言的翻译;刘江红同志负责第5到第7章的翻译;程邵斌同志负责第8到第10章的翻译;李琰同志负责第11到第13章的翻译;侯姗姗同志负责第14到第17章的翻译;黄晓晨同志负责第18到第20章及附录的翻译。

由于我们的能力有限,难免有不当之处,敬请读者不吝赐教。

译者

2007年4月

译者简介



孙家骥 北京大学信息科学技术学院教授,博士生导师,中国计算机学会高级会员。主要研究方向:计算机语言、编译系统、程序理解、软件工程。主持和参加过多项国家科技攻关项目、“863”重大项目及与国外的合作项目;近年来发表学术论文50余篇,编写出版教材、翻译学术著作10余部。曾获北京大学优秀教学奖、机电部“七五”科技攻关先进个人荣誉证书。

前言

本书第3版的出版目的与前两版相同,即介绍计算科学理论,从而为计算机专业的初级和中级学生提供计算机科学理论的健全的数学表示。促使我对上一版进行更新的原因有三个:通过提供附加的动机介绍和例子来增强表达效果;扩大知识范围,特别是在计算复杂性这一领域;为教师讲授计算机科学理论的课程提供更多的灵活性。

许多面向应用领域的学生质疑学习理论知识的重要性。然而,正是这门课强调了计算机科学问题的宏观视图。现在的编程语言和计算机体系结构很陈旧,另外,与此相关的大家感兴趣的问题都已经找到了相应的解决方案,本书中的很多内容都与这些相关。什么类型的模式可以用某种算法识别?语言如何被形式化地定义和分析?算法计算的固有能力和局限性是什么?有哪些问题的解决方法需要的时间和空间过多以至现实中无法实现?如何比较两个问题相对的困难程度?上述问题在本书中都将逐一论述。

组织

由于绝大多数本科计算机科学专业的学生都没有或缺少足够的抽象数学的基础知识,因此本书不仅介绍计算机科学的基础,而且希望能够提高学生的数学论证能力。我们通过概念的严格表述,以及使用通俗的例子来解释一些定理,来达到上述目的。每章最后都有一些练习,从而强化和加深对本章内容的理解。

为了便于大家学习,我们假设大家都没有特殊的数学预备知识。因而,第1章介绍计算理论的数学工具:基本集合理论、递归定义以及数学归纳证明。除了1.3节和1.4节介绍的特殊内容外,第1章和第2章介绍了覆盖全书的基础知识。1.3节介绍基数和对角化。它们被用在构造不确定语言和不可计算方法的存在性证明中。1.4节检查反证法中的自引用的使用。这种技术用在不确定性证明中,包括证明停机问题无解。对于那些已经学完离散数学课程的学生,第1章的大部分内容可以看作是对这些知识的复习。

考虑到计算基础的不同课程可以强调不同的主题,本书的表述和组织方式设计为允许一门课程深入探究某些特殊主题,同时提供一种加强对主要主题的研究能力,这种能力通过介绍和深入探讨计算机科学理论研究范围的材料来获取。课程的核心内容集中于形式化的自动机语言理论的经典表示,可计算性和不确定性、计算复杂性,以及作为编程语言定义和编译器设计基础的形式化语言,详见下面表格。小节后面的星号表示这部分内容可以略过但并不影响整本书表述的连贯性。带星号的小节通常包括应用的表示、相关主题的介绍或是主题中一个复杂结论的详细证明。

形式语言和自动机理论	计算理论	计算复杂性	编程语言的形式语言
第1章:1-3, 6-8	第1章:所有	第1章:所有	第1章:1-3, 6-8
第2章:1-3, 4*	第2章:1-3, 4*	第2章:1-3, 4*	第2章:1-4
第3章:1-3, 4*	第5章:1-6, 7*	第5章:1-4, 5-7*	第3章:1-4
第4章:1-5, 6*, 7	第8章:1-7, 8*	第8章:1-7, 8*	第4章:1-5, 6*, 7
第5章:1-6, 7*	第9章:1-5, 6*	第9章:1-4, 5-6*	第5章:1-6, 7*
第6章:1-5, 6*	第10章:1	第11章:1-4, 5*	第7章:1-3, 4-5*
第7章:1-5	第11章:所有	第14章:1-4, 5-7*	第18章:所有
第8章:1-7, 8*	第12章:所有	第15章:所有	第19章:所有
第9章:1-5, 6*	第13章:所有	第16章:1-6, 7*	第20章:所有
第10章:所有		第17章:所有	

形式语言和自动机理论的经典表述考察了文法和乔姆斯基体系中抽象机之间的关系。本书同时也描述了确定型有限自动机、下推自动机、线性无界自动机和图灵机的可计算性性质。抽象机的计算能力的分析,是通过使用图灵机和无限制文法产生的语言,来识别语言的等价性而建立的。

可计算性理论考察了算法问题解决的能力和局限。可计算性包括确定性和丘吉尔—图灵理论。其中,后者通过建立图灵可计算性和 μ -递归函数的等价性而获得支持。对角化证明用来证明图灵机的停机问题是无解的,而问题化简则用来构造有关算法计算的一系列问题的不确定性。

对于计算复杂性的研究,我们首先考虑计算所需要的资源的度量方法。我们选取图灵机作为评估复杂性的框架,时间和空间的复杂性是通过图灵机计算中使用的转换数量和内存数量来计算的。 \mathcal{P} 类问题可以使用确定型图灵机在多项式时间内解决,这类问题被认为是具有有效算法解决方案的一类问题。在这之后, \mathcal{NP} 类问题和NP完全理论也会给与介绍。逼近算法用来获得NP完全优化问题的近似最优解。

形式化语言理论在计算科学中最重要的应用是使用文法来指定编程语言的语法。这门课程强调使用形式化技术来定义编程语言和提出有效的分解策略,它起始于用来生成语言的上下文无关文法和用于识别模式的有限自动机的介绍。介绍语言的定义之后,第18章至第20章考察了LL和LR文法以及通过这些类型的文法定义的语言的确定性分解的性质。

练习

掌握计算科学的理论基础的过程不是一场只需旁观的体育运动会。一个人只有通过解决问题,并考察主要结论的证明,才能够充分理解理论的概念、算法和精妙之处的。也就是说,对整体理解需要更多的细微努力。为了达到这一目的,我们在每章后面都编写了一些练习。这些练习包括很多方面,既有本章介绍的主题的构造练习,也有对理论的扩展。

每部分的练习中都有几个是带星号的,这些问题要比本章的其他问题更难,在本质上更理论些,或者更特别和有趣。

符号

计算科学理论是有效性计算的能力和缺陷的数学检查。与其他形式化分析类似,这种表示必须能够提供概念、结构和操作的精确且无二义性的定义。下面的表示法将贯穿本书。

条 目	描 述	例 子
元素和字符串	字符是字母表中的小写斜体字母	a, b, abc
函数	小写斜体字母	f, g, h
集合和关系	大写字母	X, Y, Z, Σ, Γ
文法	大写字母	G, G_1, G_2
文法的变量	大写斜体字母	A, B, C, S
抽象机	大写字母	M, M_1, M_2

使用罗马字母表示集合和数学结构在一定程度上不太标准,然而这样做能使得一种结构的构成成分很容易地被识别出来。例如,上下文无关文法的结构是 $G = (\Sigma, V, P, S)$ 。仅仅从字体上我们就可以看出, G 包含三个集合和变量 S 。

整本书使用三种计数系统,每章、节、条目都给出一条引用。一个计数序列记录了定义、引理、定理、推论和算法;另一个序列用来识别例子;图、表以及练习使用章节序号来标识。

证明的结尾以■标记结束,例子的结尾以□标记结束。符号索引,包括这些符号的介绍,以及它们出现的页数^①都在附录I中给出。

① 指英文原书的页码,而非本书页码。——编辑注

补充

我们给出的部分练习的解答，仅仅是为了辅助教师授课[○]。

声明

首先，我要感谢我的妻子 Janice 和女儿 Elizabeth。由于她们的善良、耐心和顾全大局，才使我能成功地完成本书。我还要感谢 the Institut de Recherche en Informatique de Toulouse 的同事和朋友。这本书修订的第一稿是在 2004 年我访问 IRIT 期间完成的。特别要感谢 Didier Dubois 和 Henri Prade 的慷慨和好客。

每一版我都会多感谢一些对本书作出贡献的人。恳切感谢所有使用这本书的学生和教师，以及那些提出批评、校勘、修改与建议，从而促进我完善本书的人。其中很多意见我已经加入到这一版中。谢谢你们花费时间把意见发给我，希望你们以后会仍然如此。我的信箱是 tsudkamp@cs.wright.edu。

这本书的不同版本曾经得到许多计算机专家的评论，其中包括下面这些教授：Andrew Astromoff (San Francisco State University)，Dan Cooke (University of Texas-EI Paso)，Thomas Fernandez，Sandeep Gupta (Arizona State University)，Raymond Gumb (University of Massachusetts-Lowell)，Thomas F. Hain (University of South Alabama)，Michael Harrison (University of California at Berkeley)，David Hemmendinger (Union College)，Steve Homer (Boston University)，Dan Jurca (California State University-Hayward)，Klaus Kaiser (University of Houston)，C. Kim (University of Oklahoma)，D. T. Lee (Northwestern University)，Karen Lemone (Worcester Polytechnic Institute)，C. L. Liu (University of Illinois at Urbana-Champaign)，Richard J. Lorentz (California State University-Northridge)，Fletcher R. Norris (The University of North Carolina at Wilmington)，Jeffery Shallit (University of Waterloo)，Frank Stomp (Wayne State University)，William Ward (University of South Alabama)，Dan Ventura (Brigham Young University)，Charles Wallace (Michigan Technological University)，Kenneth Williams (Western Michigan University) 和 Hsu-Chun Yen (Iowa State University)，在此感谢你们。

我还要感谢 Addison-Wesley 出版公司计算机科学教育分公司和 Windfall Software 中参与这个项目的小组成员的帮助。

Thomas A. Sudkamp
Dayton, Ohio

○ 需要练习解答的教师请按书后的教师服务沟通表中列出的联系方式联系培生教育出版集团北京代表处

绪 论

计算机科学理论起源于那些有助于促进科学进步的问题：how 和 what。回答了这两个问题之后，接着是促进许多经济决策的问题：how much。这本书的目的就是解释这些问题对于学习计算机科学的重要性，并尽可能地提供答案。

形式语言理论最初是由“语言是如何定义的”这个问题引出的。在尝试捕捉自然语言的结构和各种自然语言之间的细微区别的过程中，语言学家 Noam Chomsky 提出了一个称为文法的形式化系统，它用来定义和产生文法正确的句子。大约在同时，计算机科学家正在定义编程语言的确定性和无二义性的问题。于是，这两方面的研究相互交汇就产生了编程语言 ALGOL 的文法，这种语言使用的是一种等价于上下文无关文法的形式定义。

对可计算性这一领域的探索源于两个基本问题：“什么是算法？”和“算法计算的能力和局限性是什么？”。第一个问题的回答需要计算的形式化模型。计算机和高级编程语言的结合，可以清晰地构成一个计算系统，这似乎能为可计算性的研究提供理想的框架。但是我们还需要考虑到是方法的困难程度。使用什么样的计算机？它应该有多大的存储器？使用什么样编程语言？另外，选择一种特殊的计算机或语言可能会对第二个问题的回答带来意外的、不希望的影响。在一台已配置好的计算机上能够解决的问题，并不一定也能够在另一台计算机上解决。

一个问题是否是算法可解的，有一个算法解或者没有，应该独立于使用的计算模型。因此，我们需要一个能够进行所有可能的算法计算的系统来恰当地描述可计算性问题。通用算法计算的特点自二十世纪三十年代以来就成为了数学家和逻辑学家研究的一个主要领域了。其间，大家提出了很多不同的系统作为计算模型，包括递归函数、Alonzo 的 lambda 演算、Markov 系统和 Alan Turing 设计的抽象机。所有这些系统，以及其他以此为目的地设计的系统，都能解决同样的问题集。在第 11 章中，我们给出了丘奇-图灵机的一种解释，即：一个问题有算法解当且仅当它可以被任何上述系统解决。

正因为它简单，而且其构成成分与当今的计算机相似，所以我们使用图灵机作为研究计算机的框架。图灵机与计算机有很多相似之处：它接收输入、写入内存并产生输出。虽然与计算机相比，图灵机的指令还很基本，但是，如果适当定义图灵机的指令序列，它就可以模拟计算机。尽管如此，图灵机类型还是避免了传统计算机的物理限制，即：不存在计算的时间和内存数量的上限。因此，任何计算机解决的问题都可以用图灵机解决，但反之则不然。

接受图灵机作为有限计算的统一模型之后，我们开始强调“什么是算法计算的能力和局限性？”丘奇-图灵机理论保证一个问题是可解决的，当且仅当能够设计一个合适的图灵机来解决它。于是，证明一个问题无解便简化成，无法设计一个图灵机来解决这个问题。第 12 章使用这种方法来证明几个有关我们预测计算结果的能力的重要问题是无解的。

一旦某个问题确认是可解的，那么人们就开始考虑解决这个问题时的效率和优化了。How much 这个问题初始化了计算复杂性的研究。图灵机再次提供了一个无偏颇的平台，用来比较不同问题的资源需求。图灵机的时间复杂性度量了计算需要的指令数。时间复杂性把可解决问题划分成两类：易处理的和难处理的。如果一个问题可以被图灵机解决，并且计算的执行指令数是输入的多项式函数，那么这个问题就被认为是易处理的；而不能在多项式时间内解决的问题被认为是难处理的，因为即使去解决这个问题的简单实例也需要大量的计算资源。

图灵机不但是我们考虑的惟一抽象机，而且它还是能力逐渐增强的一系列机器中的最顶尖的一个。有效计算的分析源于确定型有限自动机的性质研究。确定型有限自动机每读入一次，就要根据机器的状态和输入待处理的字符决定执行哪一条指令。尽管确定型有限自动机的状态很简单，但是它在很多方面都有应用，包括模式识别、开关电路设计和编程语言的词法分析。

另一种功能更强大的机器族，称为下推自动机，通过给有限自动机增加外部栈的存储来实现。下推自动机中增加的栈扩展了有限自动机的计算能力。由于图灵机的存在，我们对于可计算性的研究将会体现两种语言族的计算能力的特色。 [2]

语言定义和可计算性是这本书的两个主题。它们并非是计算机科学理论广阔领域中不相关的两个课题。相反，它们不可避免地纠缠在了一起。机器的可计算性可以用来识别语言；如果一个字符串在计算过程中被证明是语法正确的，那么这个字符串就可以被相应的机器接收。因此，每台机器都有相应的语言，这种语言是由被这台机器接收的字符串构成的集合。每一族抽象机的计算能力都被这族机器接收的语言来标识。记住了这一点，我们就可以深入地探讨语言的定义和有效计算的相关话题了。 [3]

目 录

出版者的话	
专家指导委员会	
译者序	
前言	
绪论	

第一部分 基 础

第1章 数学预备知识	2
1.1 集合论	2
1.2 笛卡儿积、关系和函数	4
1.3 等价关系	6
1.4 可数集合和不可数集合	7
1.5 对角化和自反	9
1.6 递归定义	11
1.7 数学归纳	13
1.8 有向图	15
1.9 练习	18
参考文献注释	20
第2章 语言	21
2.1 字符串和语言	21
2.2 语言的有穷规格说明	23
2.3 正则集合和表达式	25
2.4 正则表达式和文本搜索	28
2.5 练习	30
参考文献注释	32

第二部分 文法、自动机和语言

第3章 上下文无关文法	34
3.1 上下文无关文法和语言	36
3.2 文法和语言的例子	41
3.3 正则文法	44
3.4 验证文法	45
3.5 最左推导和二义性	48
3.6 上下文无关文法和编程语言定义	51
3.7 练习	53
参考文献注释	56

第4章 上下文无关文法范式	57
4.1 文法转换	57
4.2 消去 λ 规则	58
4.3 去掉链规则	62
4.4 无用符	64
4.5 乔姆斯基范式	67
4.6 CYK 算法	69
4.7 去掉直接左递归	71
4.8 格立巴赫范式	73
4.9 练习	77
参考文献注释	80
第5章 有限自动机	81
5.1 一个有限状态自动机	81
5.2 确定型有限自动机	82
5.3 状态图和例子	84
5.4 非确定型有限自动机	88
5.5 λ -转换	91
5.6 去掉非确定性	94
5.7 DFA 的最小化	99
5.8 练习	103
参考文献注释	107
第6章 正则语言的性质	108
6.1 有限状态机接收正则语言	108
6.2 表达式图	109
6.3 正则文法和有限自动机	111
6.4 正则语言的封闭性质	114
6.5 非正则语言	115
6.6 规则语言的泵引理	116
6.7 Myhill-Nerode 定理	119
6.8 练习	122
参考文献注释	125
第7章 下推自动机和上下文无关语言	126
7.1 下推自动机	126
7.2 PDA 的变种	129
7.3 上下文无关语言的接收	132
7.4 上下文无关语言的泵引理	136
7.5 上下文无关语言的封闭性	138

7.6 练习	140
参考文献注释	143

第三部分 可 计 算 性

第 8 章 图灵机	146
8.1 标准图灵机	146
8.2 作为语言接收器的图灵机	148
8.3 可供选择接收标准	150
8.4 多道图灵机	151
8.5 双向图灵机	151
8.6 多带图灵机	153
8.7 非确定型图灵机	157
8.8 用来枚举语言的图灵机	162
8.9 练习	166
参考文献注释	169
第 9 章 图灵可计算函数	170
9.1 函数的计算	170
9.2 数值计算	172
9.3 图灵机的顺序操作	174
9.4 函数的合成	178
9.5 不可计算函数	180
9.6 关于编程语言	181
9.7 练习	184
参考文献注释	186
第 10 章 乔姆斯基层次	187
10.1 无限制文法	187
10.2 上下文有关文法	191
10.3 线性有界自动机	192
10.4 乔姆斯基层次	195
10.5 练习	195
参考文献注释	197
第 11 章 判定问题与丘奇—图灵论题	198
11.1 判定问题的描述	198
11.2 判定问题和递归语言	199
11.3 问题归约	201
11.4 丘奇—图灵论题	203
11.5 通用机	204
11.6 练习	207
参考文献注释	208
第 12 章 不可判定性	209
12.1 图灵机的停机问题	209

12.2 问题归约和不可判定性	211
12.3 其他的停机问题	213
12.4 莱斯定理	215
12.5 不可解决的词问题	216
12.6 波斯特对应问题	218
12.7 上下文无关文法中的不可判定问题	221
12.8 练习	223
参考文献注释	225
第 13 章 μ -递归函数	226
13.1 原始递归函数	226
13.2 一些原始递归函数	228
13.3 有界操作符	230
13.4 除法函数	234
13.5 歌德尔数字和串值递归	235
13.6 可计算部分函数	237
13.7 图灵可计算函数和 μ -递归函数	240
13.8 修订的丘奇—图灵论题	243
13.9 练习	245
参考文献注释	249

第四部分 计算复杂性

第 14 章 时间复杂性	252
14.1 复杂性度量	252
14.2 增长的速度	253
14.3 图灵机的时间复杂性	256
14.4 复杂性和图灵机的变种	259
14.5 线性加速	260
14.6 语言时间复杂性的属性	262
14.7 计算机计算的模拟	266
14.8 练习	268
参考文献注释	270
第 15 章 \mathcal{P} 、 \mathcal{NP} 和库克定理	271
15.1 非确定型图灵机的时间复杂性	271
15.2 \mathcal{P} 类和 \mathcal{NP} 类	272
15.3 问题表示和复杂性	273
15.4 判定问题和复杂性类	275
15.5 哈密尔顿回路问题	276
15.6 多项式时间归约	278
15.7 $\mathcal{P} = \mathcal{NP}$?	279
15.8 可满足性问题	280
15.9 复杂类的关系	287

15.10 练习	287	18.6 练习	332
参考文献注释	289	参考文献注释	333
第 16 章 NP-完全问题	290	第 19 章 LL(k) 文法	334
16.1 归约和 NP-完全问题	290	19.1 上下文无关文法中的预读	334
16.2 三元可满足性问题	291	19.2 FIRST 集合、FOLLOW 集合和预读 集合	336
16.3 三元可满足性的归约	292	19.3 强 LL(k) 语法	338
16.4 归约和子问题	299	19.4 FIRST _{k} 集合的构造	339
16.5 最优化问题	302	19.5 FOLLOW _{k} 集合的构造	341
16.6 近似算法	303	19.6 强 LL(1) 文法	342
16.7 近似方案	305	19.7 强 LL(k) 分析器	344
16.8 练习	307	19.8 LL(k) 文法	345
参考文献注释	308	19.9 练习	346
第 17 章 其他复杂性类	309	参考文献注释	348
17.1 派生的复杂性类	309	第 20 章 LR(k) 文法	349
17.2 空间复杂性	310	20.1 LR(0) 上下文	349
17.3 空间复杂性和时间复杂性的关系	312	20.2 LR(0) 分析器	351
17.4 \mathcal{P} -空间, \mathcal{NP} -空间和萨维奇 定理	315	20.3 LR(0) 机	352
17.5 \mathcal{P} -空间完全性	318	20.4 被 LR(0) 机接收	356
17.6 一个难解问题	320	20.5 LR(1) 文法	360
17.7 练习	321	20.6 练习	365
参考文献注释	321	参考文献注释	366
第五部分 确定型语法分析		附录 I 标记索引	367
第 18 章 语法分析引论	324	附录 II 希腊字母表	370
18.1 文法图	324	附录 III ASC II 字符集	371
18.2 自顶向下语法分析	325	附录 IV Java 的 BNF 范式定义	372
18.3 归约和自底向上语法分析	328	参考文献	379
18.4 自底向上语法分析器	329	索引	384
18.5 语法分析和编译	331		

第一部分

基 础

理论计算机科学包括语言定义、模式识别、算法计算的能力和局限性、问题的复杂性分析以及它们的求解研究。这些问题都是以集合论和离散数学为基础的。第1章复习形式语言理论和计算理论中需要的数学概念、操作和表示符号。

形式语言理论源于语言学、数理逻辑和计算机科学。第2章我们将给出语言的集合论定义。这个定义很充分，因此这个定义涵盖了自然（口头和书面）语言和形式语言，但是这种通用性是以牺牲提供生成语言字符串的有效方法为代价的。为了克服这个缺点，递归定义和集合操作用来给出语言的有限规约说明。接着，我们介绍正则集合，这是出现在自动机理论、形式语言理论、开关电路和神经网络中的语言族。在第2章的结尾还给出一个使用正则表达式的实例——正则集合的简化表示——用来描述文本搜索的模式。

第1章 数学预备知识

集合论和离散数学为形式语言理论、可计算性理论和计算复杂性分析提供了数学基础。我们首先回顾集合论的表示和基本操作。集合的基数度量集合的大小，并提供无穷集合大小的准确定义。德国数学家 George Cantor 深入研究集合的属性后得出一条有趣的结论，就是存在不同大小的无穷集。尽管 Cantor 的工作仅仅表明存在一个完整的无穷集合规模层次，但是这已经足够支持我们把无穷集合分成两类的目的了。这两类分别是可数的和不可数的。如果集合的元素数目与自然数一样多，那么这个集合是可数的无穷集。如果元素数目比自然数多，就是不可数无穷集。

在本章中，我们将使用对角化论证（diagonalization argument）结构来证明定义在自然数集合上的函数集合是不可数无穷集。我们在有效过程（effective procedure）和可计算函数（computable function）的意义上达成共识后（这也是本书第三部分的主要目的），将能够确定可以用算法计算的函数集合的大小。通过比较这两个集合的大小，就可以证明存在这样的函数，它们的值不能使用任何算法过程计算得到。

[7]

一个集合可能由任意一组对象组成，我们对那些机械化生成元素的集合感兴趣。然后，我们介绍可以产生集合元素的递归定义；接着构造递归生成的集合与数学归纳法之间的关系。归纳已经被证明能够为递归产生的无穷集合中的元素性质提供一个通用的证明技巧。

在本章的最后，我们将复习有向图和树等知识，这是贯穿本书的两种结构，并以图形方式的解释了形式语言理论和计算理论的概念。

1.1 集合论

我们假设读者熟悉初等集合论的表示。在这节中，我们主要回顾这个理论的概念和记号。符号 \in 表示成员资格； $x \in X$ 表示 x 是集合 X 的成员或元素。带斜线的符号表示否， $x \notin X$ 即表示 x 不是 X 的一个成员。如果两个集合包含相同的成员，那么这两个集合相等。在本书当中，我们使用大写字母表示集合。特别地， X 、 Y 和 Z 用来表示任意集合。集合的元素用斜体表示。例如， a 、 b 、 A 、 B 、 $aaaa$ 和 abc 的字符和字符串形式表示集合的元素。

括号 $\{\}$ 用来给出集合的定义。集合中成员个数很少的时候，可以直接给出集合的定义，也就是把集合的元素列出来。集合

$$X = \{1, 2, 3\}$$

$$Y = \{a, b, c, d, e\}$$

就是使用显式的方式定义的。如果集合有大量有限个元素，或无穷数目的元素时，就必须使用隐含方式给出定义。集合的隐式定义是通过描述集合中元素的指定条件给出的。例如，包含所有的完全平方的集合定义为

$$\{n \mid n = m^2, m \text{ 为任意自然数}\},$$

隐式定义中竖线，读作“满足”。整个定义读作“ n 的集合，满足 n 是某个自然数 m 的平方”。

上面的例子中提及了自然数集合（natural numbers）。这个重要的集合，用 N 表示，包括 $0, 1, 2, 3, \dots$ 。空集（empty set）用 \emptyset 表示，这个集合里没有成员，也可以显式定义为 $\emptyset = \{\}$ 。

集合由它的全体成员完全确定。定义中元素的表示顺序并不重要。显式定义

$$X = \{1, 2, 3\}, Y = \{2, 1, 3\}, Z = \{1, 3, 2, 2, 2\}$$

表示同一个集合。 Z 的定义中包含数字 2 的多个实例。集合中的重复定义并不影响其成员资格。集合相等要求集合具有相同的成员，如上面这个例子，集合 X 、 Y 和 Z 都有自然数 1、2 和 3 作为成员。

如果 Y 的每个成员都是 X 的成员, 集合 Y 是集合 X 的**子集**, 记作 $Y \subseteq X$. 空集是任何集合的子集. 每个集合 X 都是它自身的子集. 如果 Y 是 X 的子集, 并且 $Y \neq X$, 那么 Y 是 X 的**真子集**. X 的所有子集的集合, 称作 X 的**幂集**, 记作 $\mathcal{P}(X)$. [8]

例 1.1.1 已知 $X = \{1, 2, 3\}$, X 的子集包括

$$\emptyset \quad \{1\} \quad \{2\} \quad \{3\} \\ \{1, 2\} \quad \{2, 3\} \quad \{3, 1\} \quad \{1, 2, 3\}.$$

□

集合操作使用已知的集合来创造新集合. 两个集合的**并** (union) 定义为

$$X \cup Y = \{z \mid z \in X \text{ 或者 } z \in Y\}.$$

或者是指**相容**, 表示如果 z 是 X 的成员, 或者是 Y 的成员, 或者是两者的成员, 那么 z 是 $X \cup Y$ 的成员. 两个集合的**交** (intersection) 是包含两个集合共同成员的集合, 其定义为

$$X \cap Y = \{z \mid z \in X \text{ 并且 } z \in Y\}.$$

交集为空的两个集合称为**不相交** (disjoint). n 个集合 X_1, X_2, \dots, X_n 的并集和交集分别定义为

$$\bigcup_{i=1}^n X_i = X_1 \cup X_2 \cup \dots \cup X_n = \{x \mid x \in X_i, \text{ 其中 } i = 1, 2, \dots, n\} \\ \bigcap_{i=1}^n X_i = X_1 \cap X_2 \cap \dots \cap X_n = \{x \mid x \in X_i, \text{ 其中 } i = 1, 2, \dots, n\},$$

如果

$$\text{i) } X = \bigcup_{i=1}^n X_i$$

$$\text{ii) } X_i \cap X_j = \emptyset, \text{ 其中 } 1 \leq i, j \leq n, \text{ 并且 } i \neq j.$$

那么集合 X 的子集 X_1, X_2, \dots, X_n 称为是 X 的**划分** (partition). 例如, 偶数自然数的集合 (0 被认为是偶数) 和奇数自然数的集合是自然数 \mathbb{N} 的划分.

集合 X 和集合 Y 的**差** (difference) $X - Y$, 包含属于 X 但不属于 Y 的元素:

$$X - Y = \{z \mid z \in X \text{ 并且 } z \notin Y\}.$$

设 X 是全集 U 的子集. X 对于 U 的**补集** (complement) 是包含属于 U 但不属于 X 的元素的集合. 换句话说, X 对 U 的补集是 $U - X$. 已知全集 U , X 对于 U 的补集记作 \bar{X} . 下面的恒等式叫做**德摩根定律** (DeMorgan's Laws). 它展示当 X 和 Y 是集合 U 的子集, 并且对 U 取补时的并、交和补关系. [9]

$$\text{i) } \overline{(X \cup Y)} = \bar{X} \cap \bar{Y}$$

$$\text{ii) } \overline{(X \cap Y)} = \bar{X} \cup \bar{Y}.$$

例 1.1.2 已知 $X = \{0, 1, 2, 3\}$, $Y = \{2, 3, 4, 5\}$, \bar{X} 和 \bar{Y} 分别表示 X 和 Y 对于 U 的补集. 那么

$$\begin{aligned} X \cup Y &= \{0, 1, 2, 3, 4, 5\} & \bar{X} &= \{n \mid n > 3\} \\ X \cap Y &= \{2, 3\} & \bar{Y} &= \{0, 1\} \cup \{n \mid n > 5\} \\ X - Y &= \{0, 1\} & \bar{X} \cap \bar{Y} &= \{n \mid n > 5\} \\ Y - X &= \{4, 5\} & \overline{(X \cup Y)} &= \{n \mid n > 5\} \end{aligned}$$

右边一列的最后两个集合显示了德摩根定律中的等式. □

子集的定义提供了证明 X 是 Y 的子集的方法: 我们必须证明 X 的每一个元素也是 Y 的元素. 当 X 是有限集时, 我们可以直接检查 X 中的每个元素, 看它是否是 Y 的元素. 当 X 包含无穷多个元素时, 则需要使用别的方法来解决. 解决的策略就是证明 X 中的任意元素都属于 Y .

例 1.1.3 我们要证明 $X = \{8n - 1 \mid n > 0\}$ 是 $Y = \{2m + 1 \mid m \text{ 是奇数}\}$ 的子集. 为了更好地理解集合 X 和 Y , 我们可以列出 X 和 Y 的元素.

$$X: 8 \cdot 1 - 1 = 7, 8 \cdot 2 - 1 = 15, 8 \cdot 3 - 1 = 23, 8 \cdot 4 - 1 = 31, \dots$$

$$Y: 2 \cdot 1 + 1 = 3, 2 \cdot 3 + 1 = 7, 2 \cdot 5 + 1 = 11, 2 \cdot 7 + 1 = 13, \dots$$

为了建立这种相容关系, 我们必须证明 X 的每个元素也是 Y 的元素. X 中的任意元素 x 都有 $8n - 1$ 的形式, 其中, $n > 0$. 设 $m = 4n - 1$, 那么 m 是奇数, 并且

$$\begin{aligned}
 2m+1 &= 2(4n-1)+1 \\
 &= 8n-2+1 \\
 &= 8n-1 \\
 &= x.
 \end{aligned}$$

[10] 因此, x 也属于 Y , 并且 $X \subseteq Y$. □

集合相等可以通过集合相容来定义. 如果 $X \subseteq Y$ 并且 $Y \subseteq X$, 那么集合 X 和 Y 相等. 这样就简单地证明了 X 的每个元素都是 Y 的元素, 反之亦然. 当构造两个集合相等关系时, 通常分别证明这两个相容关系, 然后再将它们结合起来, 从而证明集合的相等.

例 1.1.4 证明集合

$$\begin{aligned}
 X &= \{n \mid n = m^2 \text{ 任意自然数 } m > 0\} \\
 Y &= \{n^2 + 2n + 1 \mid n \geq 0\}
 \end{aligned}$$

相等. 首先, 我们要证明 X 的每个元素都是 Y 的元素. 设 $x \in X$, 则存在某个自然数 $m > 0$ 使得 $x = m^2$. 设 m_0 就是这个数, 则 x 可以写成

$$\begin{aligned}
 x &= (m_0)^2 \\
 &= (m_0 - 1 + 1)^2 \\
 &= (m_0 - 1)^2 + 2(m_0 - 1) + 1.
 \end{aligned}$$

设 $n = m_0 - 1$, 那么我们就可以得到 $x = n^2 + 2n + 1$, 其中 $n \geq 0$. 因此, x 是集合 Y 的元素.

然后, 我们构造另一个相容关系. 设 $y = n_0^2 + 2n_0 + 1$ 是 Y 的元素, 因式分解得到 $y = (n_0 + 1)^2$. 因此, y 是某个自然数的平方且大于 0, 因此它也是 X 的元素. 因为 $X \subseteq Y$ 并且 $Y \subseteq X$, 所以 $X = Y$. □

1.2 笛卡儿积、关系和函数

笛卡儿积 (cartesian product) 是建立两个已知集合的元素的有序对的集合的操作. 集合 X 和 Y 的笛卡儿积, 表示成 $X \times Y$, 可以定义为

$$X \times Y = \{[x, y] \mid x \in X \text{ 并且 } y \in Y\}.$$

X 和 Y 的二元关系 (binary relation) 是 $X \times Y$ 的子集. 自然数的排序可以用来产生基于集合 $N \times N$ 的关系 LT (少于). 这个关系是 $N \times N$ 的子集, 定义为

$$LT = \{[i, j] \mid i < j \text{ 并且 } i, j \in N\}.$$

[11] 符号 $[i, j] \in LT$ 表示 i 小于 j . 例如, $[0, 1], [0, 2] \in LT$ 并且 $[1, 1] \notin LT$.

笛卡儿积可以进一步被泛化, 从而能让我们使用有限数目的集合来构造新的集合. 如果 x_1, x_2, \dots, x_n 是 n 个元素, 那么 $[x_1, x_2, \dots, x_n]$ 就称为有序的 n 项 (ordered n -tuple). 一个有序对是有序的 2 项的另一个名字. 有序的 3 项、4 项和 5 项分别指的是三元组、四元组和五元组. n 个集合 X_1, X_2, \dots, X_n 的笛卡儿积定义为

$$X_1 \times X_2 \times \dots \times X_n = \{[x_1, x_2, \dots, x_n] \mid x_i \in X_i, \text{ 其中 } i = 1, 2, \dots, n\}.$$

X_1, X_2, \dots, X_n 上的 n 元关系是 $X_1 \times X_2 \times \dots \times X_n$ 的子集. 1 元、2 元和 3 元关系分别称为一元的、二元的和三元的.

例 1.2.1 已知 $X = \{1, 2, 3\}$, $Y = \{a, b\}$. 那么

a) $X \times Y = \{[1, a], [1, b], [2, a], [2, b], [3, a], [3, b]\}$

b) $Y \times X = \{[a, 1], [a, 2], [a, 3], [b, 1], [b, 2], [b, 3]\}$

c) $Y \times Y = \{[a, a], [a, b], [b, a], [b, b]\}$

d) $X \times Y \times Y = \{[1, a, a], [1, b, a], [2, a, a], [2, b, a], [3, a, a], [3, b, a],$

$[1, a, b], [1, b, b], [2, a, b], [2, b, b], [3, a, b], [3, b, b]\}$ □

非形式化地说,从集合 X 到集合 Y 的函数 (function) 就是从 X 的元素到 Y 的元素的映射,并且 X 中的每个元素最多映射到 Y 中的一个元素。从 X 到 Y 的函数可以表示成 $f: X \rightarrow Y$ 。 Y 的元素通过函数 f 赋值给元素 $x \in X$ 来获得,记作 $f(x)$ 。集合 X 为函数的定义域 (domain), X 的元素是函数 f 的操作数。 f 的值域 (range) 是通过赋值给 X 而获得的 Y 的子集。因此,函数 $f: X \rightarrow Y$ 的值域是集合 $\{y \in Y \mid y = f(x) \text{ 对于任意 } x \in X\}$ 。

给每个人赋以年龄的关系就是从人到自然数的函数。注意,值域中的每个元素可能会被赋给定义域中多个元素——因为有很多人的年龄相同。而且,并不是所有的自然数都是函数的值域,自然数 1000 就不可能赋给任何人。

函数的定义域是集合,但是这个集合经常是两个或多个集合的笛卡儿积。函数

$$f: X_1 \times X_2 \times \cdots \times X_n \rightarrow Y$$

就是 n 个变量的函数 (n -variable function) 或操作。变量 x_1, x_2, \dots, x_n 的函数值表示成 $f(x_1, x_2, \dots, x_n)$ 。具有一个变量、两个变量或者三个变量的函数经常用来指一元操作、二元操作和三元操作。函数 $sq: \mathbb{N} \rightarrow \mathbb{N}$, 把 n 赋给每个自然数,这就是一元操作。当函数的定义域包含集合 X 和它自身的笛卡儿积的时候,这个函数就称为是 X 上的一元操作。加法和乘法就是 \mathbb{N} 上的二元操作。

12

函数 f 为定义域的成员和 f 值域的成员建立了关联。函数的本质定义就是依赖于这种关系的。从 X 到 Y 的全函数 (total function) f 就是基于 $X \times Y$ 的二元关系的,并且要求它满足下面两个性质:

i) 对于每个 $x \in X$, 都存在 $y \in Y$ 使得 $[x, y] \in f$;

ii) 如果 $[x, y_1] \in f$, 并且 $[x, y_2] \in f$, 那么 $y_1 = y_2$ 。

第一个性质保证 X 中的每个元素都被赋给 Y 中的一个成员,因此才有全这个术语。第二个性质保证这个赋值是惟一的。前面定义的 LT 关系不是全函数,因为它不满足第二个条件。 $\mathbb{N} \times \mathbb{N}$ 上表示大于的关系不能满足上述的任何一个性质,这是为什么呢?

例 1.2.2 已知 $X = \{1, 2, 3\}$ 和 $Y = \{a, b\}$ 。从 X 到 Y 的八个全函数如下所示。

x	$f(x)$	x	$f(x)$	x	$f(x)$	x	$f(x)$
1	a	1	a	1	a	1	b
2	a	2	a	2	b	2	a
3	a	3	b	3	a	3	a
x	$f(x)$	x	$f(x)$	x	$f(x)$	x	$f(x)$
1	a	1	b	1	b	1	b
2	b	2	a	2	b	2	b
3	b	3	b	3	a	3	b

□

从 X 到 Y 的部分函数 (partial function) f 是 $X \times Y$ 上的关系,并且在这个关系中,如果 $[x, y_1] \in f$ 且 $[x, y_2] \in f$, 那么 $y_1 = y_2$ 。如果存在 $y \in Y$ 使得 $[x, y] \in f$, 那么 f 就是关于 x 的部分函数。否则, f 就没有给出 x 的定义。全函数也是一种部分函数,只不过它为定义域的所有元素给出了定义。

尽管我们使用关系的形式给出了函数的定义,但是我们将使用标准形式 $f(x) = y$ 来表示 y 的值是通过函数 f 为 x 赋值而得到的值,即 $[x, y] \in f$ 。符号 $f(x)^\wedge$ 表示不存在为参数 x 定义的部分函数 f , 符号 $f(x)^\downarrow$ 用来表示定义的 $f(x)$ 并不显式地给出它的值。

整数除法定义 f 从 $\mathbb{N} \times \mathbb{N}$ 到 \mathbb{N} 的二元部分函数 div 。用 i 除以 j 获得结果就是商,记为 $div(i, j)$ 。例如, $div(3, 2) = 1$, $div(4, 2) = 2$, $div(1, 2) = 0$ 。使用前面的表示,对所有的 j (除了 0) 有 $div(i, 0)^\uparrow$ 和 $div(i, j)^\downarrow$ 。

如果 X 的每个元素都映射到值域中的惟一个元素, 那么全函数 $f: X \rightarrow Y$ 就是一对一的 (one-to-one) 形式化地, 如果 $x_1 \neq x_2$, 则有 $f(x_1) \neq f(x_2)$, 那么 f 是一对一的。如果 f 的值域是全集 Y , 那么函数 $f: X \rightarrow Y$ 就是满射 (onto)。全函数在定义域和值域之间的映射关系既是一对一的, 又是满射的。

例 1.2.3 函数 f, g 和 s 是定义在 \mathbf{N} 到 $\mathbf{N} - \{0\}$ 上的, 后者是正整数的集合。

i) $f(n) = 2n + 1$

ii) $g(n) = \begin{cases} 1, & \text{如果 } n = 0 \\ n, & \text{其他} \end{cases}$

iii) $s(n) = n + 1$

函数 f 是一对一的映射, 但不是满射。 f 的值域包含奇数。 g 定义的从 \mathbf{N} 到 $\mathbf{N} - \{0\}$ 的映射显然是满射, 但不是一对一的映射, 因为 $g(0) = g(1) = 1$ 。函数 s 既是一对一的映射, 又是满射, 因此它定义了一个自然数到它的后继之间的映射。□

例 1.2.4 在前面的例子中, 我们注意到函数 $f(n) = 2n + 1$ 是一对一的映射, 但不是映射到整个集合 $\mathbf{N} - \{0\}$ 上的。尽管如此, 从 \mathbf{N} 到奇自然数的集合的映射既是一对一的映射, 也是满射的。我们将使用 f 来演示如何证明函数具有这些性质。

一对一: 为了证明函数是一对一的, 我们要证明当 $f(n) = f(m)$ 时, n 和 m 一定相等。假设 $f(n) = f(m)$, 那么

$$\begin{aligned} 2n + 1 &= 2m + 1 && \text{或者} \\ 2n &= 2m, && \text{最终,} \\ n &= m. \end{aligned}$$

接着, $n \neq m$ 意味着 $f(n) \neq f(m)$, 因此 f 是一对一的。

满射: 为了构造 f 把 \mathbf{N} 映射到整个奇自然数的集合, 我们必须证明每个奇自然数都在 f 的值域中。如果 m 是奇自然数, 那么它就可以写成 $m = 2n + 1$, 其中 $n \in \mathbf{N}$ 。于是 $f(n) = 2n + 1 = m$ 并且 m 位于 f 的值域中。□

1.3 等价关系

我们曾经用笛卡儿积 $X \times X$ 的子集来形式化地定义集合 X 上的二元关系。非形式化地, 我们使用关系来表示集合的两个元素之间是否存在某种性质。如果有序对的元素满足前面所说的条件, 那么它就在该关系之中。例如, 属性少于定义了自然数集合上的二元关系。使用这个属性定义的关系是集合 $\mathbf{LT} = \{[i, j] \mid i < j\}$ 。

中缀表达式经常被用来表示常见二元关系中的成员资格。在这种标准使用当中, $i < j$ 意味着 i 小于 j , 并且 $[i, j]$ 在上面定义的关系 \mathbf{LT} 中。

我们现在考虑一种称为等价的关系, 这种关系可以用于划分潜在集合。等价关系通常使用中缀表达式 $a \equiv b$ 来表示 a 等价于 b 。

定义 1.3.1 集合 X 上的二元关系 \equiv 是等价关系, 除非它满足下面的条件

i) 自反性: $a \equiv a$, 对于所有的 $a \in X$ 。

ii) 对称性: $a \equiv b$ 意味着 $b \equiv a$, 对于所有 $a, b \in X$ 。

iii) 传递性: $a \equiv b$ 并且 $b \equiv c$, 则意味着 $a \equiv c$, 对于所有 $a, b, c \in X$ 。

定义 1.3.2 已知 \equiv 是 X 上的等价关系。元素 $a \in X$ 的等价类 (equivalence class) 使用关系 \equiv 定义了集合 $[a]_{\equiv} = \{b \in X \mid a \equiv b\}$ 。

例 1.3.1 已知 \equiv_p 是 \mathbf{N} 上使用 $n \equiv_p m$ 定义的奇偶关系, 当且仅当 n 和 m 具有同样的奇偶性 (奇数或偶数)。为了证明 \equiv_p 是等价关系, 我们必须证明它的对称性、自反性和传递性。

i) 自反性: 对于每个自然数 n , n 和它本身有相同的奇偶性, 并且 $n \equiv_p n$ 。

ii) 对称性: 如果 $n \equiv_p m$, 那么 n 和 m 有相同的奇偶性, 并且 $m \equiv_p n$ 。

iii) 传递性: 如果 $n \equiv_p m$ 并且 $m \equiv_p k$, 那么 n 和 m 有相同的奇偶性, 并且 m 和 k 有相同的奇偶性。然后 n 和 k 有相同的奇偶性, 并且 $n \equiv_p k$ 。

奇偶关系的两个等价类 \equiv_p 是 $[0]_{\equiv_p} = \{0, 2, 4, \dots\}$ 和 $[1]_{\equiv_p} = \{1, 3, 5, \dots\}$ 。□

等价类通常写成 $[a]_{\equiv}$ ，其中 a 是这个类中的一个元素。在上面的这个例子中， $[0]_{\equiv_p}$ 用来表示偶自然数的集合。引理 1.3.3 证明如果 $a \equiv b$ ，则 $[a]_{\equiv} = [b]_{\equiv}$ 。因此，选择哪个元素来表示这个类都是不重要的。

引理 1.3.3 已知 \equiv 是 X 上的等价关系，并且 a 和 b 都是 X 的元素。那么我们可以得到 $[a]_{\equiv} = [b]_{\equiv}$ 或者 $[a]_{\equiv} \cap [b]_{\equiv} = \emptyset$ 。

证明：假设 $[a]_{\equiv}$ 和 $[b]_{\equiv}$ 的交集不空，那么存在元素 c 属于两个等价类中。使用对称性和传递性，我们可以得到 $[b]_{\equiv} \subseteq [a]_{\equiv}$ 。因为 c 既属于 $[a]_{\equiv}$ 又属于 $[b]_{\equiv}$ ，所以我们可以得到 $a \equiv c$ 并且 $b \equiv c$ 。根据对称性有 $c \equiv b$ 。再使用传递性就能得到 $a \equiv b$ 。 [15]

现在我们假设 d 是 $[b]_{\equiv}$ 中的任意元素，那么 $b \equiv d$ 。结合 $a \equiv b$ 和 $b \equiv d$ ，并使用传递性可以得到 $a \equiv d$ 。所以 $d \in [a]_{\equiv}$ 。因此我们已经证明了 $[b]_{\equiv}$ 中的任意元素也属于 $[a]_{\equiv}$ 。根据类似的证明，我们可以构造 $[a]_{\equiv} \subseteq [b]_{\equiv}$ 。依据这两个相容关系，我们就可以得到想要的集合相等关系。■

定理 1.3.4 设 \equiv 是 X 上的等价关系。 \equiv 的等价类是 X 的划分。

证明：根据引理 1.3.3，我们知道等价类形成 X 的不相交的子集族。设 a 是 X 的任意元素。根据自反性有 $a \in [a]_{\equiv}$ 。因此 X 的每个元素都属于某个等价类。这些等价类的并集就是整个集合 X 。■

1.4 可数集合和不可数集合

集合的基数是对集合大小的度量。直观上说，集合的基数就是集合中元素的数目。当处理有限集合的时候，这个非形式化的定义就足够了。我们可以通过数集合元素的数目来获得集合的基数。但当把这种方法扩展到无穷集合的时候，就会出现明显的困难。

通过建立一对一的集合元素映射，我们可以证明两个有限集合具有相同数目的元素。例如，映射

$$\begin{aligned} a &\rightarrow 1 \\ b &\rightarrow 2 \\ c &\rightarrow 3 \end{aligned}$$

说明集合 $\{a, b, c\}$ 和 $\{1, 2, 3\}$ 具有相同数目的元素。这种使用映射的方法来比较集合的大小，对于有限集合和无穷集合都适用。

定义 1.4.1 i) 如果从 X 到 Y 之间存在一个完全一对一的满射，那么 X 和 Y 这两个集合具有相同的基数。

ii) 如果从 X 到 Y 之间存在一个完全的一对一的映射，那么集合 X 的基数不大于集合 Y 。

注意这两个定义惟一的差别就是映射覆盖集合 Y 的程度。如果一对一的映射覆盖的区域是整个 Y ，那么这两个集合具有相同的基数。

集合 X 的基数表示成 $\text{card}(X)$ 。(i)和(ii)里面的关系可以分别表示成 $\text{card}(X) = \text{card}(Y)$ 和 $\text{card}(X) \leq \text{card}(Y)$ 。如果 $\text{card}(X) \leq \text{card}(Y)$ 并且 $\text{card}(X) \neq \text{card}(Y)$ ，那么 X 的基数严格小于 Y 的，写成 $\text{card}(X) < \text{card}(Y)$ 。Schröder-Bernstein 定理为基数构造了 \leq 和 $=$ 之间的类似关系。Schröder-Bernstein 定理的证明留作练习。 [16]

定理 1.4.2 (Schröder-Bernstein) 如果 $\text{card}(X) \leq \text{card}(Y)$ 并且 $\text{card}(X) \neq \text{card}(Y)$ ，那么 $\text{card}(X) < \text{card}(Y)$ 。

有限集合的基数可以用集合中元素的数目表示。因此， $\text{card}(\{a, b\}) = 2$ 。和自然数集合的基数相同的集合称为可数无穷的 (countably infinite 或者 denumerable)。直观地，如果集合的成员可以按照某种顺序数出来，那么集合就是可数的。映射 f 建立了它和自然数之间的对应关系，并提供了相应的排序：第一个元素是 $f(0)$ ，第二个是 $f(1)$ ，第三个是 $f(2)$ 等等。术语可数 (countable) 指的是集合有限或者可以数出来。不可能数出来的集合称为不可数 (uncountable)。

集合 $\mathbb{N} - \{0\}$ 是可数无穷的；函数 $s(n) = n + 1$ 定义了从 \mathbb{N} 到 $\mathbb{N} - \{0\}$ 之间的一对一映射。看起来可能有些荒谬，集合 $\mathbb{N} - \{0\}$ 是去掉集合 \mathbb{N} 中的一个元素获得的，可是它却和集合 \mathbb{N} 有相同数目的元

二维格结构, 其中输入值是横轴, 函数值是纵轴。

考虑使用 $f(n) = f_n(n) + 1$ 定义的函数 $f: \mathbf{N} \rightarrow \mathbf{N}$ 。 f 的值通过给格的 diagonal 增加 1 来获得, 因此, 我们将其命名为对角化。 通过给每个 i 定义 $f, f(i) \neq f_i(i)$ 。 因此, f 不在序列中。 然而已知序列 f_0, f_1, f_2, \dots 包含所有的全函数, 所以矛盾。 因为假设函数的数目是可数无穷的, 从而导致了这个矛盾, 所以集合是不可数的。

对角化是一种通用的证明技巧, 用来证明集合不是可数的。 正如前面的例子所示, 使用对角化构造不可数就是反证法的一个证明依据。 第一步假设集合是可数的, 因此它的元素可以穷举地列出来。 通过产生一个不应该出现在序列中的元素从而产生矛盾。 列举元素的时候不需要附加任何条件, 除了它必须包含在集合的所有元素。 通过对角化产生的矛盾证明了不存在列举所有元素的可能, 因此, 集合是不可数的。 下面的例子将再次展示了这个技巧。

19

例 1.4.3 如果存在自然数 i 使得 $f(i) = i$, 那么从 \mathbf{N} 到 \mathbf{N} 的函数 f 就存在固定点 (fixed point)。 例如, $f(n) = n2$ 有固定点 0, 而 $f(n) = n2 + 1$ 没有固定点。 我们将证明没有固定点的函数是不可数的。 这个证明类似于证明从 \mathbf{N} 到 \mathbf{N} 的所有自然数的数目是不可数的, 除非我们现在有一个附加条件: 构造不在序列中的元素。

假设没有固定点的函数的数量是可数的。 于是, 这些函数就可以列出来, 像 f_0, f_1, f_2, \dots 。 为了根据我们的假设构造矛盾, 我们构造了一个没有固定点的函数, 并且它也不在序列中。 考虑函数 $f(n) = f_n(n) + n + 1$ 。 定义 f 中新加的 $n + 1$ 保证对于所有的 n 有 $f(n) > n$ 。 因此 f 没有固定点。 类似于上面给出的证明, 对于所有的 i 有 $f(i) \neq f_i(i)$ 。 因此, 序列 f_0, f_1, f_2, \dots 不是可以穷举的, 并且我们可以得出结论: 没有固定点的函数的数量是不可数的。 \square

例 1.4.4 $\mathcal{P}(\mathbf{N})$ 是 \mathbf{N} 的子集的集合, 且是不可数的。 假设 \mathbf{N} 的子集的集合是可数的, 那么它们就可以列举出来 N_0, N_1, N_2, \dots 。 定义 \mathbf{N} 的子集 D 如下: 对于每个自然数 j ,

$$j \in D \text{ 当且仅当 } j \notin N_j$$

根据我们的构造, 如果 $0 \notin N_0$ 则 $0 \in D$, 如果 $1 \notin N_1$ 则 $1 \in D$, 等等。 集合 D 显然是自然数的集合。 根据假设, N_0, N_1, N_2, \dots 是 \mathbf{N} 的子集的无穷列举。 因此, 对于任意 i 有 $D = N_i$ 。 数字 i 是否属于集合 D 呢? 根据 D 的定义,

$$i \in D \text{ 当且仅当 } i \notin N_i$$

但是因为 $D = N_i$, 所以就变成

$$i \in D \text{ 当且仅当 } i \notin D,$$

这是矛盾的。 因此, 我们的假设 $\mathcal{P}(\mathbf{N})$ 是可数的是错误的, 于是我们得出结论: $\mathcal{P}(\mathbf{N})$ 是不可数的。

为了进一步了解“对角化”技术, 我们考虑二维格, 它的横轴是自然数, 纵轴是集合 N_0, N_1, N_2, \dots 。 如果 $j \in N_i$, 那么行 N_i 和列 j 确定的格的位置就是是, 否则, 就是否。 考虑格的 diagonal 入口, 即数字 j 和集合 N_j 之间的关系, 我们可以构造集合 D 。 根据我们给出的定义 D 的方式, 数字 j 是集合 D 的元素当且仅当 N_j 和 j 对应的位置的入口处是否。 \square

20

1.5 对角化和自反

除了可以用于证明集合的基数之外, 对角化还提供了一种用于描述某种属性或关系是固有矛盾的方法。 这一点被用于不存在性的证明当中, 因为没有对象满足这个性质。 不存在的对角化证明经常依赖于自反性。 一对象分析它自身的行为、属性和特点。 罗素悖论、图灵机停机问题的不确定性, 以及对于数论的不确定性的 Gödel 证明都是基于和自反相关的矛盾论的。

前面小节中使用的对角化证明使用了表格中的操作, 它基于纵轴上的操作数和横轴上的参数来解释操作数和参数之间的关系。 在每个例子中, 操作数和它的参数的类型都不相同。 在自反中, 同一族的对象包含相同的操作数和参数。 我们使用理发师悖论, 罗素悖论的一个相当简单的例子, 来解释对角化和自反。

理发师悖论是有关一个神秘小镇中的谁给谁理发的问题的。 我们被告知这个小镇中的男人都可以

给自己理发，而小镇的理发师（他自己就是一个男人）只给那些不能自己理发的人理发。我们希望考虑这段陈述和这个小镇存在的可能性。在这个例子里，小镇里男人的集合既是操作数也是参数。我们或者让别人来给我们理发，或者自己理发。设 $M = \{p_1, p_2, p_3, \dots, p_i, \dots\}$ 是小镇中所有男性的集合。右图表示了理发关系的集合。从图中可以看出，如果 p_i 给 p_j 理发，那么在第 i 行、第 j 列的位置就是 1，否则就是 0。每列都有一个入口标着 1，其他的入口标着 0。每个人或者自己理发，或者被理发师理发。理发师一定是小镇中的成员之一，因此，他一定是某个 p_j 。那么表格中 i 行 j 列的位置的值是多少呢？这是一个经典的自反问题。我们会问当某一个对象既是操作者（理发的人），又是被操作者（被理发的人）时，会发生什么事情？

	p_1	p_2	p_3	\dots	p_i	\dots
p_1	-	-	-	\dots	-	\dots
p_2	-	-	-	\dots	-	\dots
p_3	-	-	-	\dots	-	\dots
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\dots
p_i	-	-	-	\dots	-	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

[21] 谁给理发师理发呢？如果理发师能够给自己理发，但是实际他不能，因为他只给不能自己理发的人理发。如果他不给自己理发，那么他就必须给自己理发因为他应该给所有不自己理发的人理发。我们证明了这个小镇的理发属性是矛盾的，因此这样的城镇是不存在的。

罗素悖论遵循了相同的模式，但是它的结论要远比神秘小镇的不存在更重要。集合论的重要原理之一就是任何可以描述出来的性质或条件都可以用来定义集合，而集合就是由满足这个条件的对象的构成，这是 19 世纪晚期 Cantor 提出的。可能没有对象满足这个性质，或者有限多个、或无穷多个对象满足这个性质。不管元素的数目和类型怎样，这些元素都构成了一个集合。罗素构造了一个基于自反性的论证，证明了这个命题不可能是正确的。

罗素悖论考察的关系是一个集合属于另一个集合的成员资格。对于每个集合 X ，我们问这样一个问题，“集合 Y 是否是集合 X 的元素？”这是一个不合理的问题，因为一个集合当然可以是另一个集合的元素。右边的表格给出了这个问题的肯定和否定例子。

X	Y	$Y \in X?$
$\{a\}$	$\{a\}$	否
$\{a, b\}$	a	是
$\{\{a\}, a, \emptyset\}$	\emptyset	是
$\{\{a, b\}, \{a\}\}$	$\{\{a\}\}$	否
$\{\{a, b\}, b, b\}$	$\{a, b\}$	是

值得注意的是，这个问题不是集合 Y 是否是集合 X 的子集，而是它是否是集合 X 的元素。

成员资格可以用右面的表格描述。其中，横轴与纵轴表示的集合。表项 $[i, j]$ 是 1 表示 X_j 是集合 X_i 的元素，而 0 表示 X_j 不是集合 X_i 的元素。

通过识别成员资格问题中的操作符和操作数，我们可以获得一个自反问题。即：我们问集合 X_i 是否是它自身的元素。前面的表格中的对角化 $[i, i]$ 包含了这个问题的回答。“ X_i 是 X_i 的元素吗？”

	X_1	X_2	X_3	\dots	X_i	\dots
X_1	-	-	-	\dots	-	\dots
X_2	-	-	-	\dots	-	\dots
X_3	-	-	-	\dots	-	\dots
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\dots
X_i	-	-	-	\dots	-	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

现在考察一个集合不是它自身的元素。根据这个性

[22] 质是否可以定义一个集合？有很多实例显然满足这个性质：集合 $\{a\}$ 不是它自身的元素。这个性质的满足通过对角化的补来体现。集合 X_i 不是它自身的元素当且仅当单元 $[i, i]$ 是 0。

假设 $S = \{X \mid X \notin X\}$ 是集合。 S 属于 S 吗？如果 S 是它自身的元素，那么根据 S 的定义它不属于 S 。而且，如果 S 不属于 S ，那么它必须属于 S ，因为它不是它自身的元素。这显然是矛盾的。由于我们的假设：满足属性 $X \notin X$ 的集合的全体构成一个集合，所以这是矛盾的。

我们给出了一个可描述的性质，但是它却不能用来定义集合。这就表明了 Cantor 关于集合的普遍性的断言是错误的。罗素悖论衍生的影响更是深远的。这样，集合论的研究就从基于简单定义的基础转移到由公理和推理规则构成的形式化系统上，并帮助开创了数学的形式主义哲学。在第 12 章我们将使用自反性来构造计算机科学理论中的重要结论：停机问题的不确定性。

1.6 递归定义

事实上,在形式语言和自动机理论中,我们感兴趣的许多集合都有无穷多个元素。因此,我们有必要寻找一种方法来描述、产生和识别属于无穷集合的元素。在前面的小节中,我们使用省略号(\dots)表示自然数的集合。这看起来合理,因为每个阅读本书的人都熟悉自然数,并且知道0,1,2,3后面是什么。然而,这个定义对于不熟悉基本的10进制系统和数学表示的人而言可能是完全不正确的。这样的人没有这样的概念,即符号4是下一个元素,而1492是自然数。

在数学理论的发展过程中,像语言理论或自动机理论,定理和证明可以使用的仅仅是相应理论的概念定义。因此,这就需要知道相应领域的对象和操作的准确定义。定义的方法必须能够使我们的朋友、陌生人或没有直觉的计算机能够产生,并“理解”集合的元素所具有的性质。

集合 X 的递归定义(recursive definition)描述了构造集合元素的方法。定义使用了两种成分:基础和操作的集合。基础包括直接指定为 X 的成员的元素的有限集合。操作用于使用以前定义的成员来构造集合的新元素。递归定义的集合 X 包括所有由集合元素使用有限步操作获得的元素。

这个递归定义集合过程中的关键词是产生。显然,没有过程可以列出自然数的完整集合。然而,任何特殊数都可以按照起始于0,并构造自然数序列的方式获得。这从本质上描述了定义自然数集合的递归过程。这个想法可以用下面的定义形式化地描述出来。

23

定义 1.6.1 自然数集合 \mathbf{N} 的递归定义是使用后继函数 s 构造的。

- i) 基础步骤: $0 \in \mathbf{N}$
- ii) 递归步骤: 如果 $n \in \mathbf{N}$, 那么 $s(n) \in \mathbf{N}$
- iii) 封闭: $n \in \mathbf{N}$ 只有当它可以由0经过有限步应用操作 s 获得。

基础步骤直接给出了0是自然数。在(ii)中,一个新的自然数通过定义过的数和后继操作给出。封闭这部分保证集合只包含那些可以由0使用后继操作获得的元素。定义1.6.1产生了无穷序列0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, \dots 。这个序列可以缩写成0, 1, 2, 3, \dots 。无论如何,可以用熟悉的阿拉伯数字描述的东西,都可以使用这种非缩写的方式来完成。

递归过程的本质是使用同一过程或结构的简单实例来定义复杂的过程或结构。在自然数的例子中,“简单”意味着小。定义1.6.1使用已定义过的数来递归地定义一个新数。

自然数被定义后,如何理解它们的性质呢?我们通常把加、减和乘的操作与自然数联系到一起。我们通过强迫,或者是记忆或是简单的重复的方式学习这些。对于不熟悉这个操作的人,或者是要进行加运算的人而言,就必须正确定义加的概念。一个人不可能记住自然数的所有可能组合,但是我们可以递归地构造一个方法来计算任何两个数的和。后继函数是引入到自然数上的惟一操作。因此,加的定义可以仅使用0和 s 。

定义 1.6.2 下面递归定义了 m 和 n 的和,递归发生在 n 上——和操作的第二个操作数。

- i) 基础步骤: 如果 $n=0$, 那么 $m+n=m$ 。
- ii) 递归步骤: $m+s(n)=s(m+n)$ 。
- iii) 封闭: $m+n=k$ 仅当这个等式可以从 $m+0=m$ 开始使用有限步递归步骤获得。

给定的域中,封闭这一步在操作的递归定义中常常被省略。在这个例子中,假设这个操作是定义在定义域的所有元素上的。上面给出的加操作就是定义在 $\mathbf{N} \times \mathbf{N}$ 的所有元素上的。

n 的后继与 m 的和,使用简单的例子来定义的,就是 m 和 n 的和的后继操作。作为递归操作对象 n 的选择是任意的;在给定 n 之后, m 的操作也可以这样定义。

24

依据定义1.6.2中给出的构造,任何两个自然数的和可以使用0和 s 来计算。其中,0和 s 是定义自然数的基本元素。例1.6.1给出了 $3+2$ 的递归计算过程。

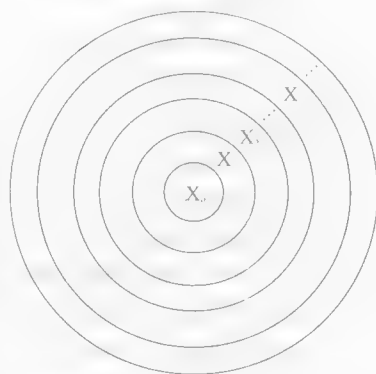
例 1.6.1 数字 3 和 2 分别是 $s(s(s(0)))$ 和 $s(s(0))$ 的缩写。它们的和递归地定义为

$$\begin{aligned} & s(s(s(0))) + s(s(0)) \\ &= s(s(s(s(0))) + s(0)) \\ &= s(s(s(s(s(0)))) + 0) \\ &= s(s(s(s(s(0))))) \quad (\text{基础}) \end{aligned}$$

最终的值就是数字 5 的表示。□

图 1-1 解释了由基础 X_0 生成集合 X 的递归过程。每个同心圆表示构造的一个步骤。 X_0 表示基础元素, 以及由它们使用一步操作获得的元素。 X_i 包括经过不超过 i 步操作获得的元素。在递归定义的生成过程中产生了一个可数的有限序列的嵌套集合。集合 X 可以看作是 X_i 的无穷并。设 x 是 X 的元素, 并且 X_i 是 x 出现的第一个集合。这就意味着 x 可以由基本元素经过 j 步操作获得。尽管 X 的每个元素都可以通过操作的有限次应用而获得, 但是应用的步骤是没有上限的。这种使用有限但是无限步骤生成的性质, 就是递归定义的重要性质。

后继操作可以用来递归地定义 $\mathbf{N} \times \mathbf{N}$ 上的关系。笛卡儿乘积 $\mathbf{N} \times \mathbf{N}$ 经常使用格上的表示有序对的点来描绘。遵循标准惯例, 横轴表示有序对的第一个元素, 纵轴表示第二个。图 1-2 (a) 的阴影包含所有的有序对 $\langle i, j \rangle$, 其中 $i < j$ 。根据 1.2 节的描述, 这个集合就是 1.2 节描述的小于关系 LT 。



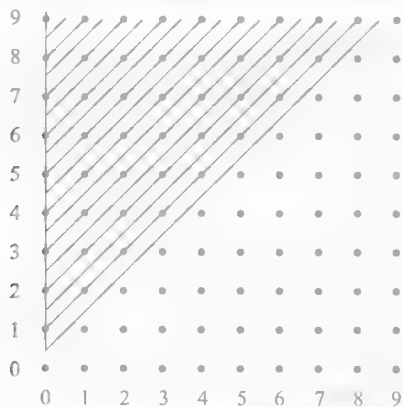
X 的递归生成:

$X_0 = \{x | x \text{ 是基础元素} \}$

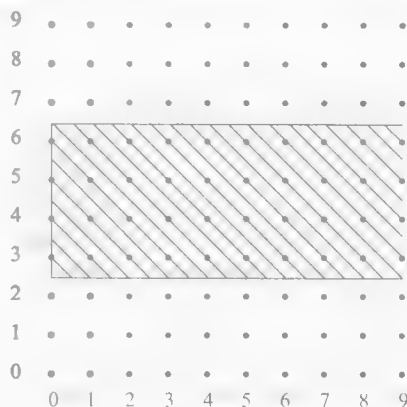
$X_{i+1} = X_i \cup \{x | x \text{ 可以通过 } i+1 \text{ 步操作生成} \}$

$X = \{x | \text{对于某些 } j \geq 0, x \in X_j \}$

图 1-1 递归定义集合的嵌套序列



(a)



(b)

图 1-2 $\mathbf{N} \times \mathbf{N}$ 上的关系

例 1.6.2 关系 LT 定义如下:

i) 基础步骤: $[0, 1] \in LT$ 。

ii) 递归步骤: 如果 $[m, n] \in LT$, 那么 $[m, s(n)] \in LT$ 并且 $[s(m), s(n)] \in LT$ 。

iii) 封闭: $[m, n] \in LT$ 只有当它可以由 $[0, 1]$ 经过有限步递归操作得到。

使用递归生成的无穷并描述, LT 的定义生成了嵌套集合的序列 LT_i , 使得

$$\begin{aligned} LT_0 &= \{[0, 1]\} \\ LT_1 &= LT_0 \cup \{[0, 2], [1, 2]\} \\ LT_2 &= LT_1 \cup \{[0, 3], [1, 3], [2, 3]\} \\ LT_3 &= LT_2 \cup \{[0, 4], [1, 4], [2, 4], [3, 4]\} \\ &\vdots \\ LT_i &= LT_{i-1} \cup \{[j, i+1] \mid j=0, 1, \dots, i\} \\ &\vdots \end{aligned}$$

□

LT 的构造说明了递归定义的元素生成过程并不一定是惟一的。有序对 $[1, 3] \in \text{LT}$, 可以通过下面两个不同的操作序列获得。

$$\begin{array}{ll} \text{基础: } [0, 1] & [0, 1] \\ 1: [0, s(1)] = [0, 2] & [s(0), s(1)] = [1, 2] \\ 2: [s(0), s(2)] = [1, 3] & [1, s(2)] = [1, 3]. \end{array}$$

26

例 1.6.3 图 1-2 (b) 的阴影区域包含了所有第二个元素是 3、4、5 或 6 的有序对。这个集合的递归定义称为 X :

- i) 基础步骤: $[0, 3], [0, 4], [0, 5]$ 和 $[0, 6]$ 都属于 X 。
 - ii) 递归步骤: 如果 $[m, n] \in X$, 那么 $[s(m), n] \in X$ 。
 - iii) 封闭: $[m, n] \in X$, 只有当它可以由基础元素经过有限步应用操作获得。
- 集合 X 的序列可以按照下面的方式递归定义来生成:

$$X_i = \{[j, 3], [j, 4], [j, 5], [j, 6] \mid j = 0, 1, \dots, i\}.$$

□

1.7 数学归纳

构造集合元素和这个集合上的操作之间的关系, 需要通过证明来验证你所假定的性质。我们如果仅仅考虑单个元素, 是无法证明性质对于无穷集合中的每个元素都是成立的。数学归纳的原理为证明一个性质对于给定集合中的每个元素都成立提供了充分的依据。归纳利用递归过程生成的嵌套集合族, 把它的性质从一些基础元素拓展到整个集合上。

27

数学归纳的原理 设 X 是由基础 X_0 通过递归定义的集合, 并且 $X_0, X_1, X_2, \dots, X_i, \dots$ 是利用递归生成的集合序列。已知 P 是 X 元素的性质。如果能够证明:

- i) P 对于集合 X_0 中的所有元素都成立,
- ii) 当 P 对于集合 $X_0, X_1, X_2, \dots, X_i, \dots$ 中的所有元素都成立, 那么 P 对于 X_{i+1} 中的每个元素也成立。

那么, 根据数学归纳的原理, P 对于 X 中的每个元素都成立。

数学归纳原理的正确性可以使用在递归定义 X 的过程中构造的集合序列来给出直观的展示。把圈 X_i 涂上阴影表示 P 对于 X_i 中的每个元素都成立。第一个条件要求内部集合涂阴影。条件 (ii) 声明阴影部分可以从任何圆圈扩展到下一个同心圈。图 1-3 解释了这个过程是如何最终把整个集合 X 涂上阴影的。

根据前面的论证, 我们可以清楚地了解到数学归纳原理的合理性。假设条件 (i) 和条件 (ii) 得到满足, 但是 P 并不能使 X 中的每个元素都成立, 这样就获得了另一个合理性。如果 P 不能对 X 中的所有元素成立, 那么至少有一个集合 X_i 不满足 P 。设 X 是第一个这样的集合。因为条件 (i) 保证 P 对于 X_0 中所有的元素成立, 所以 j 不可能是 0。现在假设我们选择 j , 使得 P 对于 X_j 中的所有元素都成立。条件 (ii) 要求 P 对于 X_i 中的所有元素都成立。这就意味着在序列中不存在第一个集合使得性质 P 不成立。因此, P 一定是对于所有的 X_i 都成立的, 于是对于 X 成立。

归纳证明包括三个步骤。第一步是证明性质 P 对于基础集合中的每个元素都成立。这对应于建立数学归纳原理的条件 (i)。第二条是归纳假设的陈述。归纳假设就是假设性质 P 对于集合 X_0, X_1, \dots, X_n 中的每个元素都成立。使用归纳假设时, 归纳步骤证明了 P 可以扩充到 X_{n+1} 的每个元素。当我们完成归纳步骤时, 数学归纳原理的条件也就全部得到了满足。于是就可以得出结论, P 对于 X 中的所有元素都成立。

在例 1.6.2 中, 我们给出了一个用以生成关系 LT 的递归定义, 在这个关系中我们考虑的就是满足 $i < j$ 的有序对 $[i, j]$ 。根据这个定义生成的每个有序对都满足这个不等关系吗? 我们将利用这个问题来解释在递归定义的集合中的归纳证明步骤。

第一步是直接说明集合中的所有元素都满足不等式。LT 递归定义的基础是集合 $\{[0, 1]\}$ 。因为 $0 < 1$, 所以归纳证明的基础步骤得到了满足。

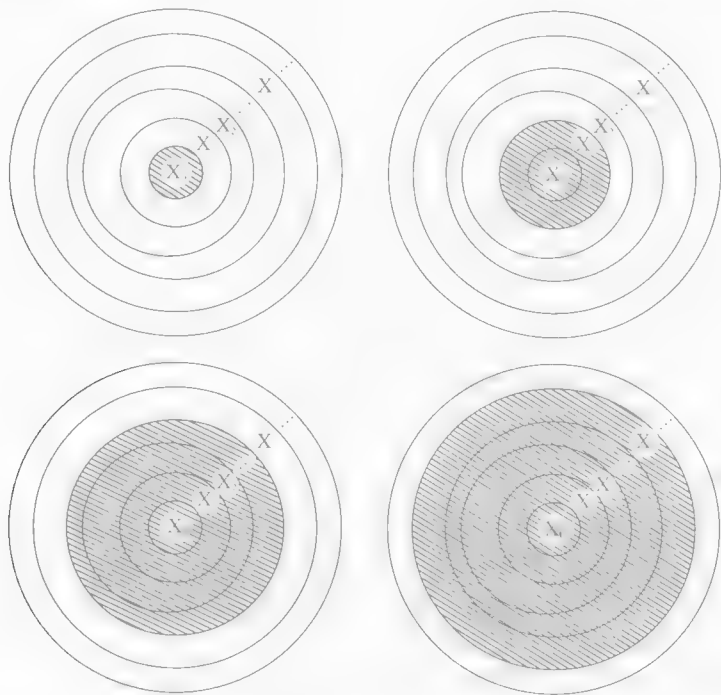


图 1-3 数学归纳的原理

归纳假设说明假设 $x < y$ 对于所有有序对 $[x, y] \in LT_n$ 都成立。在这个归纳步骤中，我们必须证明对于所有有序对 $[i, j] \in LT_{n+1}$ 来说 $i < j$ 都成立。LT 定义的递归步骤建立了集合 LT_{n+1} 和 LT_n 之间的关系。设 $[x, y]$ 是 LT_{n+1} 中的有序对，那么对于任意的 $[x, y] \in LT_n$ 都有 $[i, j] = [x, s(y)]$ 或者 $[i, j] = [s(x), s(y)]$ 。于是根据归纳假设就有 $x < y$ 。如果 $[i, j] = [x, s(y)]$ ，那么

$$i = x < y < s(y) = j.$$

类似地，如果 $[i, j] = [s(x), s(y)]$ ，则有

$$i = s(x) < s(y) = j.$$

不管哪种情况均有 $i < j$ ，并且不等关系扩展到了 LT_{n+1} 上的所有有序对。这就完成了要求的归纳证明，因此不等关系对于 LT 中的所有有序对都成立。

在证明关系 LT 中的每个有序对 $[i, j]$ 都满足 $i < j$ 时，归纳步骤仅仅使用了假设：性质对于利用递归步骤以前生成的元素都成立。这种证明有时称为简单归纳（simple induction）。当归纳步骤使用全部的归纳假设时——性质对于所有以前产生的元素都成立——称为强归纳（strong induction）。例 1.7.1 使用强归纳来建立数学表达式中操作数的数目和括号的数目之间的关系。

例 1.7.1 数学表达式的集合 E 使用符号 a, b ，操作符 $+$ 和 $-$ ，以及括号，定义如下：

i) 基础步骤： a 和 b 都属于 E。

ii) 递归步骤：如果 u 和 v 属于 E，那么 $(u+v)$ 、 $(u-v)$ 和 $(-v)$ 都属于 E。

iii) 封闭：表达式属于 E 则它可以由基础有限步应用递归步骤获得。

递归定义产生的在递归应用第一、第二和第三步中的表达式分别是 $(a+b)$ 、 $(a+(b+b))$ 和 $((a+a)-(b-a))$ 。我们使用归纳来证明表达式 u 中括号的数目是操作符数目的二倍。即： $n_p(u) = 2n_o(u)$ ，其中， $n_p(u)$ 表示 u 中括号的数目，而 $n_o(u)$ 表示操作符的个数。

基础步骤：包含表达式 a 和 b 的归纳的基础。在这里， $n_p(a) = 0 = 2n_o(a)$ ，并且 $n_p(b) = 2n_o(b)$ 。

归纳假设：假设不超过 n 次使用递归步骤得到的表达式都满足 $n_p(u) = 2n_o(u)$ ，那么 u 总属于 E_n 。

归纳步骤：设 w 是 $n+1$ 次应用递归步骤获得的表达式。那么 $w = (u+v)$ 、 $w = (u-v)$ 或者 $w = (-v)$ ，其中， u 和 v 都是 E_n 中的表达式。根据归纳假设，

$$\begin{aligned}n_p(u) &= 2n_o(u) \\ n_p(v) &= 2n_o(v).\end{aligned}$$

如果 $w = (u+v)$ 或者 $w = (u-v)$

$$\begin{aligned}n_p(w) &= n_p(u) + n_p(v) + 2 \\ n_o(w) &= n_o(u) + n_o(v) + 1.\end{aligned}$$

[30]

因此,

$$2n_o(w) = 2n_o(u) + 2n_o(v) + 2 = n_p(u) + n_p(v) + 2 = n_p(w).$$

如果 $w = (-v)$, 那么

$$2n_o(w) = 2(n_o(v) + 1) = 2n_o(v) + 2 = n_p(v) + 2 = n_p(w).$$

因此性质 $n_p(w) = 2n_o(w)$ 对于所有的 $w \in E_n$ 都成立。于是, 我们根据数学归纳得出结论, 对于 E 中的所有表达式都是正确的。□

归纳证明经常使用自然数作为潜在的递归定义集合。递归定义的集合使用的是定义 1.6.1 中给出的基础 0: 第 n 步应用递归步骤产生了自然数 n , 相应的归纳步骤把性质推广, 使之从 $0, \dots, n$ 到 $n+1$ 都得到满足。

例 1.7.2 用数学归纳法来证明 $0+1+\dots+n = n(n+1)/2$ 。使用和式, 我们可以把前面的表达式记成

$$\sum_{i=0}^n i = n(n+1)/2.$$

基础步骤: 基础步骤是 $n=0$ 。关系是通过计算目标等式两边的值来直接构造的

$$\sum_{i=0}^0 i = 0 = 0(0+1)/2.$$

归纳假设: 假设对于所有的值 $k=1, 2, \dots, n$ 有

$$\sum_{i=0}^k i = k(k+1)/2.$$

归纳步骤: 我们需要证明

$$\sum_{i=0}^{n+1} i = (n+1)(n+1+1)/2 = (n+1)(n+2)/2.$$

[31]

归纳假设构造了包含 n 个或数目更少的整数的序列的和来作为结果, 与归纳假设相结合, 我们可以得到

$$\begin{aligned}\sum_{i=0}^{n+1} i &= \sum_{i=0}^n i + (n+1) && (+ \text{ 的联合}) \\ &= n(n+1)/2 + (n+1) && (\text{归纳假设}) \\ &= (n+1)(n/2+1) && (\text{分配性质}) \\ &= (n+1)(n+2)/2.\end{aligned}$$

既然我们可以构造数学归纳原理的条件, 那我们就可以得出结论, 即对于所有的自然数都成立。□

证明的每一步都必须遵循前面操作数和归纳假设中构造的性质。归纳证明的策略是使用公式涵盖应用于简单例子的性质实例。完成这些之后, 就可以使用归纳假设了。使用归纳假设之后, 剩下的证明部分经常包括使用一些代数操作来构造目标结果。

1.8 有向图

数学结构包含集合、集合中的可区分元素和集合上的函数和关系。可区分元素 (distinguished element) 指的是这个集合中的具有区别于其他元素的性质的元素。根据定义 1.6.1, 自然数可以表示成 $(\mathbb{N}, s, 0)$ 的结构。集合 \mathbb{N} 包含自然数, s 是 \mathbb{N} 上的一元函数, 0 是 \mathbb{N} 中的可区分元素。0 之所以是可区分的, 是因为它在自然数定义中的重要作用。

图经常被用来描绘图中的数学实体的本质特点, 这种图可以辅助人们直观地理解这些概念。从形式上而言, 有向图 (directed graph) 是包含集合 \mathbb{N} 和它上面的二元关系 A 的数学结构。 \mathbb{N} 的元素称为

图的节点 (node) 或顶点 (vertices), 而 A 的元素称为弧 (arc) 或边 (edge) 关系 A 是邻接 (adjacency) 关系。若 $x, y \in A$, 则节点 y 是邻接 x 的。有向图中从 x 到 y 的弧用从 x 到 y 的带箭头的直线表示。在这个带箭头的图中, y 就是弧的头, 而 x 是弧尾。节点 x 的入度 (in-degree) 是以 x 为头的弧的数目。 x 的出度 (out-degree) 是以它为尾的弧的数目。图 1-4 中节点 a 的入度为 2、出度是 1。

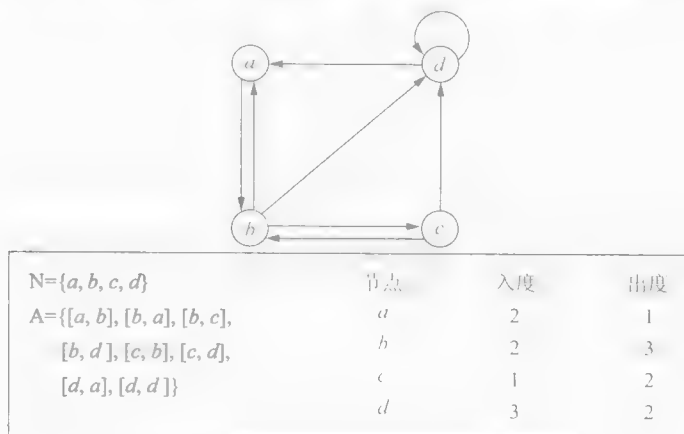


图 1-4 有向图

32

有向图 $G=(N, A)$ 中从节点 x 到节点 y 的路径 (path) 是节点和弧的序列 $x_0, [x_0, x_1], x_1, [x_1, x_2], x_2, \dots, x_{n-1}, [x_{n-1}, x_n], x_n$ 且满足 $x=x_0, y=x_n$ 。节点 x 是路径的初始节点, y 是终点。路径上的每对节点 x_i, x_{i+1} 通过弧 $[x_i, x_{i+1}]$ 相连。路径的长度就是路径上弧的数目。我们经常使用弧的序列来描述路径。

从任何节点到它自身都有一条长度为 0 的路径, 这种路径称为空路径 (null path)。起始于同一节点的长度不小于 1 的路径称为回路 (cycle)。如果一个回路不包含任何子回路, 那么这个回路是简单的。图 1-4 中的路径 $a, b, [b, c], c, d, [d, a]$ 就是一个长度为 4 的简单回路。至少包含一个回路的有向图称为有环图 (cyclic)。没有回路的图称为无环图 (acyclic)。

有向图中的弧不仅包含节点的邻接。带标记的有向图是 (N, L, A) 结构, 其中 L 是标记的集合, A 是 $N \times N \times L$ 上的关系。元素 $[x, y, v] \in A$ 是从 x 到 y 的标记为 v 的弧。弧上的标记记录着相邻节点的关系。图 1-5 上的标记给出了从 Chicago 到 Minneapolis、Seattle、San Francisco、Dallas 和 St. Louis 以及回 Chicago 的旅程的距离。



图 1-5 带标记的有向图

一个有序树 (ordered tree) 或者仅是一棵树, 就是一个无环有向图, 并且它的每个节点都有惟一条从树的根 (root) 到该节点的路径。根也是一种特殊的节点。根的入度为0, 而其他节点的入度都是1。树的结构是 (N, A, r) , 其中, N 是节点的集合, A 是邻接关系, 而 $r \in N$ 是树的根。计算机科学中的术语“树”既包含了家族树 (Family Tree) 的概念, 又包含了树作为植物的本质的概念。尽管树也是有向图, 但是弧上的箭头通常在树的图形表示中被省略。图 1-6 (a) 给出根为 x_1 的树 T 。

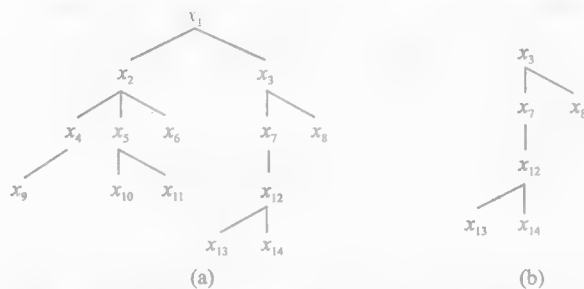


图 1-6 (a) 根为 x_1 的树; (b) x_3 生成的子树

如果 y 邻接于 x , 节点 y 叫做节点 x 的子节点 (child), x 是 y 的父节点。与邻接关系相伴的概念是任何节点的子节点的顺序。当描绘一棵树时, 我们通常按照从左到右的顺序列出节点的子节点。 x_1 在 T 上的子节点按顺序是 x_4, x_5 和 x_6 。

出度为0的节点称为叶子 (leaf)。所有其他的节点都是中间节点。根的深度 (depth) 是0, 任何其他节点深度都是这些节点的父节点深度加一。树的高度或深度就是树中的节点的深度的最大值。

如果从 x 到 y 存在一条路径, 那么节点 y 称作节点 x 的后代 (descendant), x 是 y 的祖先 (ancestor)。根据这个定义, 每个节点都是它自身的后代和祖先。祖先和后代的关系可以使用相邻关系 (练习 43 和练习 44) 来递归地定义。两个节点 x 和 y 的最小共同祖先 (minimal common ancestor) 是它们共同祖先中的一个, 并且是这些共同祖先的后代。在图 1-6 (a) 的树种, x_{10} 和 x_6 的最小共同祖先是 x_5 , 而 x_{10} 和 x_6 的最小共同祖先是 x_2 , x_{10} 和 x_{14} 的是 x_1 。

树 T 的子树是 T 的子图, 并且根据子树自身的特点决定它也是棵树。节点 x 后代的集合和这个集合的邻接关系的限制构成了以 x 为根的子树。这个树就叫做 x 生成的子树。

树中兄弟的排序可以扩展到 $N \times N$ 上的关系 LEFTOF。LEFTOF 试图获取树的图中每个节点左边节点的性质。对于两个节点 x 和 y , 它们中的任何一个都不是另一个的祖先。关系 LEFTOF 是根据节点的最小共同祖先生成的子树来定义的。设 z 是 x 和 y 的最小共同祖先, 并且 z_1, z_2, \dots, z_n 顺次是 z 的子节点。那么 x 就是由 z 的子节点生成的子树, 记作 z_i 。类似地, 对于某个 j , y 是 z 生成的子树。因为 z 是 x 和 y 的最小共同祖先, $i \neq j$ 。如果 $i < j$, 那么 $x, y \in \text{LEFTOF}$; 否则 $y, x \in \text{LEFTOF}$ 。根据这个定义, 没有节点是 LEFTOF 的祖先。如果 x_{10} 在 x_{13} 的左边, 那么 x_{10} 就必须在 x_5 的左边, 因为它们都是它们父节点的第一个子节点。这种存在于一个祖先的左边或右边的现象是这个图的特点之一, 而不是节点排序的性质。

关系 LEFTOF 可以用来给树中的叶子排序。树的边界 (frontier) 应该从叶子开始, 按照 LEFTOF 关系的顺序构造。 T 的边界是序列 $x_9, x_{10}, x_{11}, x_6, x_{13}, x_{14}, x_8$ 。

当一族图递归地被定义后, 数学归纳的原理就可以被用来证明这些性质对整个族的所有图都成立了。我们将使用归纳来演示严格二叉树每个节点有一个叶子或者有两个子节点构成的图中叶的数目和弧的数目的关系。

例 1.8.1 每个节点至多有两个子节点的树称作二叉树 (binary tree)。如果每个节点都是叶子, 或者正好有两个子节点, 那么这样的树叫做严格二叉。严格二叉树的家族可以递归地定义如下:

i) 基础步骤: 有向图 $T = (\{r\}, \emptyset, r)$ 是个严格二叉树。

ii) 递归步骤: 如果 $T_1 = (N_1, A_1, r_1)$ 和 $T_2 = (N_2, A_2, r_2)$ 都是严格二叉树, 其中 N_1 和 N_2 不相连, 并且 $r \notin N_1 \cup N_2$, 那么

$$T = (N_1 \cup N_2 \cup \{r\}, A_1 \cup A_2 \cup \{[r, r_1], [r, r_2]\}, r)$$

是个严格二叉树。

iii) 封闭: 仅当 T 能够从基本元素开始, 应用有限的递归步骤构造出来, T 才是一个严格二叉树

严格二叉树或者是一个单个节点, 或者是在两个严格独立的二叉树中间添加根和弧来构成。设 $lv(T)$ 和 $arc(T)$ 表示严格二叉树中叶和弧的数目。我们通过归纳法证明了对于所有的严格二叉树都有 $2lv(T) - 2 = arc(T)$ 。

基础步骤: 基础步骤包括严格二叉树, 形如 $(\{r\}, \emptyset, r)$ 。在这个例子中, 等式明显成立, 因为这种形式的树有一个叶节点, 而且没有弧。

归纳假设: 假设使用不超过 n 步应用递归步骤生成的每个严格二叉树 T 都满足 $2lv(T) - 2 = arc(T)$ 。

归纳步骤: 设 T 是经过 $n+1$ 步应用递归步骤生成的严格二叉树。 T 是由节点 r 和两个以前构造的严格二叉树 T_1 和 T_2 构造的, 它们的根分别是 r_1 和 r_2 。

节点 r 不是叶节点, 因为它有到 T_1 和 T_2 根节点构成的弧。因此, $lv(T) = lv(T_1) + lv(T_2)$ 。 T 的弧包括它的构成树的弧和从 r 开始的两条弧。

因为 T_1 和 T_2 都是经过不超过 n 步应用递归步骤而生成的严格二叉树, 所以我们能应用归纳假设构造目标等式。根据归纳假设,

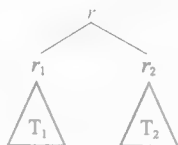
$$2lv(T_1) - 2 = arc(T_1)$$

$$2lv(T_2) - 2 = arc(T_2).$$

这样,

$$\begin{aligned} arc(T) &= arc(T_1) + arc(T_2) + 2 \\ &= 2lv(T_1) - 2 + 2lv(T_2) - 2 + 2 \\ &= 2(lv(T_1) + lv(T_2)) - 2 \\ &= 2(lv(T)) - 2. \end{aligned}$$

□



1.9 练习

1. 已知 $X = \{1, 2, 3, 4\}$, $Y = \{0, 2, 4, 6\}$ 。给出下面集合的显式定义。

- | | |
|---------------------|---------------|
| a) $X \cup Y$ | b) $X \cap Y$ |
| c) $X - Y$ | d) $Y - X$ |
| e) $\mathcal{P}(X)$ | |

2. 已知 $X = \{a, b, c\}$, $Y = \{1, 2\}$ 。

- 列出 X 的所有子集
- 列出 $X \times Y$ 的成员
- 列出从 Y 到 X 的所有全函数

3. 已知 $X = \{3^n \mid n > 0\}$, $Y = \{3n \mid n \geq 0\}$ 。证明 $X \subseteq Y$ 。

4. 已知 $X = \{n^3 + 3n^2 + 3n \mid n \geq 0\}$, $Y = \{n^3 - 1 \mid n > 0\}$ 。证明 $X = Y$ 。

5. 证明德摩根定律。使用集合相等的定义来构造恒等式。

6. 给出满足下面条件的函数 $f: \mathbf{N} \rightarrow \mathbf{N}$ 。

- f 是全函数且为一对一的映射, 但不是满射。
- f 是全函数和满射, 但不是一对一的映射。
- f 是全函数、一对一的映射和满射, 但不是恒等式。
- f 不是全函数, 但是满射。

7. 证明: 使用 $f(n) = n^2 + 1$ 定义的函数 $f: \mathbf{N} \rightarrow \mathbf{N}$ 是一对一的映射, 但不是满射。

8. 已知 $f: \mathbf{R}^+ \rightarrow \mathbf{R}^+$ 是使用 $f(x) = 1/x$ 定义的函数, 其中, \mathbf{R}^+ 表示正实数的集合. 证明 f 是一对一的映射, 但不是满射.
9. 给出分别满足下面条件的 $\mathbf{N} \times \mathbf{N}$ 上的二元关系.
 - a) 自反、对称、但不传递.
 - b) 自反、传递、但不对称.
 - c) 对称、传递、但自反.
10. 已知当且仅当 $n = m$ 时, $=$ 是 \mathbf{N} 上根据 $n \equiv m$ 定义的二元关系. 证明 $=$ 是等价关系, 并且描述 $=$ 的等价类.
11. 对于所有的 $n, m \in \mathbf{N}$, 根据 $n \equiv m$ 定义 \mathbf{N} 上的二元关系 \equiv . 证明 \equiv 是等价关系, 并描述 \equiv 的等价类.
12. 证明二元关系 LT (小于) 不是等价关系.
13. 已知如果 $n \bmod p = m \bmod p$, 则可以根据 $n \equiv_p m$ 定义 \mathbf{N} 上的二元关系 \equiv_p . 对于 $p \geq 2$, 证明 \equiv_p 是等价关系. 描述 \equiv_p 的等价类.
14. 已知 X_1, \dots, X_i 是集合 X 的划分. 请定义 X 上的等价关系 \sim , 其中 X 的等价类就是集合 X_1, \dots, X_i .
15. 二元关系 \sim 是按照下面关系定义的自然数的有序对: $(m, n) \sim (j, k)$, 当且仅当 $m + k = n + j$. 证明 \sim 是 $\mathbf{N} \times \mathbf{N}$ 上的等价关系.
16. 证明偶自然数的集合是可数的.
17. 证明偶整数的集合是可数的.
18. 证明非负有理数的集合是可数的.
19. 证明两个不相交可数集合的交也是可数的.
20. 证明从 \mathbf{N} 到 $\{0, 1\}$ 存在不可数个全函数.
21. 如果对于任意的 $n \in \mathbf{N}$ 都有 $f(n) = f(n+1)$, 则从 \mathbf{N} 到 \mathbf{N} 的全函数称作重复 (repeating), 否则, f 就是不重复的. 证明存在数量不可数的重复函数, 并且证明存在数量也不可数的非重复函数.
22. 如果对于所有的 $n \in \mathbf{N}$ 都有 $f(n) < f(n+1)$, 则称从 \mathbf{N} 到 \mathbf{N} 的全函数 f 是单调增加的 (monotone increasing). 证明存在不可数的单调增加函数.
23. 证明从 \mathbf{N} 到 \mathbf{N} 的映射中存在着不可数个具有固定点的全函数. 固定点的定义参照例 1.4.3.
24. 如果对于每个 n 都有 $f(n) = n - 1$, n 或 $n + 1$, 那么就称从 \mathbf{N} 到 \mathbf{N} 的全函数 f 是几乎恒等的 (nearly identity). 证明存在不可数的几乎恒等函数.
25. 证明区间 $[0, 1]$ 中的实数集合是不可数的. 提示: 在实数的十进制扩展中使用对角化论证. 确定每个自然数都使用惟一的无穷十进制扩展表示.
26. 已知 F 是形如 $f: [0, 1] \rightarrow \mathbf{N}$ (从 $[0, 1]$ 映射到自然数的函数) 的全函数集合. 这些函数的集合是可数的还是不可数的? 证明你的结论.
27. 证明: 使用 $X \sim Y$ 定义的集合上的二元关系是成立的, 当且仅当 $\text{card}(X) = \text{card}(Y)$ 是等价关系.
28. 证明 Schröder-Bernstein 定理.
29. 使用操作数 s 为 $\mathbf{N} \times \mathbf{N}$ 上的等于关系构造递归定义.
30. 使用后继操作符 s 为 $\mathbf{N} \times \mathbf{N}$ 的大于关系构造递归定义.
31. 在 $\mathbf{N} \times \mathbf{N}$ 上, 为直线 $n = 3m$ 上的点 (m, n) 构成的集合给出递归定义. 用 s 作为定义中的操作符.
32. 给出一个点 (m, n) 的集合的递归定义, 要求这个点位于 $\mathbf{N} \times \mathbf{N}$ 的线 $n = 3m$ 上或在这条直线的下侧, 并且在定义中使用 s 作为操作数.
33. 使用操作数 s 和加法递归地定义自然数的乘法操作.
34. 使用操作数 s 递归地定义下面的操作.

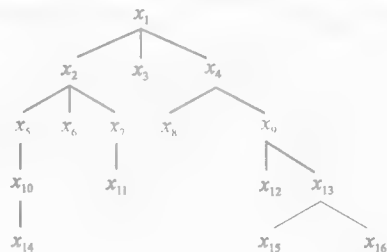
$$\text{pred}(n) = \begin{cases} 0, & \text{如果 } n = 0 \\ n - 1, & \text{其他} \end{cases}$$

35. 自然数集合上的减法定义为

$$n - m = \begin{cases} n - m, & \text{如果 } n > m \\ 0, & \text{其他} \end{cases}$$

这个操作称作是真减 (proper subtraction)。使用操作数 s 和 $pred$ 来递归地定义真减。

36. 已知 X 是有限集合。给出 X 的子集的集合的递归定义, 并在定义中使用并操作。
37. 给出 \mathbf{N} 的有限子集的集合的递归定义, 并在定义中使用并和后继 s 。
38. 证明: 对于所有的 $n > 0$, 都有 $2 + 5 + 8 + \cdots + (3n - 1) = n(3n + 1)/2$ 。
39. 证明: 对于所有的 $n \geq 0$, 都有 $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ 。
40. 证明: 对于所有的 $n > 2$, 都有 $1 + 2^n < 3^n$ 。
41. 证明: 对于所有的 $n \geq 0$, 3 都是 $n^3 - n + 3$ 的因子。
42. 已知 $P = \langle A, B \rangle$ 是由两个命题字母 (布尔变量) 构成的。 P 上结构良好的合取的或析取的布尔表达式构成的集合 E 由下面的形式定义:
- 基础步骤: $A, B \in E$ 。
 - 递归步骤: 如果 $u, v \in E$, 则 $(u \vee v) \in E$ 并且 $(u \wedge v) \in E$ 。
 - 封闭: 一个表达式属于 E 仅当从基础步骤开始经过有限步迭代而获得这个表达式。
- 直接给出集合 E_0, E_1 和 E_2 上的布尔表达式。
 - 对 E 中每个布尔表达式使用数学归纳证明: 命题字母出现的次数多于操作数的数目。对于表达式 u , 用 $n_p(u)$ 表示 u 中的命题字母的数目, $n_o(u)$ 表示 u 中操作数的数目。
 - 使用数学归纳法证明: 对于 E 中的每个布尔表达式, 左括号的数目等于右括号。
43. 给出有向图中给定节点 x 可通过某条路径到达的所有节点的递归定义。在这个定义中可以使用邻接关系。这个定义同时也给出了树的节点的后代的集合。
44. 递归地定义树的节点 x 的后继的集合。
45. 列出图 1-6 (a) 的树中的 LEFTOF 关系的成员。
46. 使用下面的树, 给出 (a) 到 (e) 的值。
- 树的深度
 - x_{11} 的后继
 - x_{14} 和 x_{11} , x_{15} 和 x_{11} 的最小共同后继
 - x_2 产生的子树
 - 树的边界
47. 证明: 有 n 个叶子的严格二叉树包含 $2n - 1$ 个节点。
48. 深度为 n 的完全二叉树 (complete binary tree) 也是严格二叉树, 并且 $1, 2, \dots, n - 1$ 层上的每个节点都是父节点, 同时 n 层上的节点也都是叶节点。证明深度为 n 的完全二叉树有 $2^{n+1} - 1$ 个节点。



参考文献注释

本章介绍的内容是离散数学第一课通常会涵盖的内容。关于离散数学结构对于计算机科学的基础的重要作用的全面介绍, 可以参考 Bobrow 和 Arbib[1974]。

还有其他一些经典书籍提供了本章介绍的内容的细节。集合论的介绍可以参考 Halmos[1974]、Stoll[1961] 以及 Fraenkel、Bar-Hillel 和 Levy[1984]。后者首先给出了在集合论中产生的罗素悖论和其他悖论的精辟描述。对角化论证最早由 Cantor 在 1874 年提出, 并在 Cantor[1947] 中得以重新介绍。Wilson[1985]、Ore[1961]、Bondy 和 Murty[1977] 以及 Busacker 和 Saaty[1965] 中的内容介绍了图论, 归纳、递归和它们在理论计算机科学中的关系在 Wand[1980] 中有所介绍。

第2章 语 言

语言这一概念包括大量看上去截然不同的种类,如自然语言、计算机语言和数学语言。通用的语言定义就要囊括所有这些类型的语言。本章我们给出了一个纯粹集合论意义上的语言的定义:语言是字母表上的字符串的集合。字母表是语言的符号的集合,字母表上的字符串是字母表的符号的有限序列。

尽管字符串固有的结构很简单,但是它们对于交流和计算的重要作用却是怎么说都不足为过的。句子“The sun did not shine”就是由英语单词构成的字符串。英语的字母表是单词和停顿符号的集合,它们都出现在句子当中。数学等式

$$p = (n \times r \times t) / v$$

是由变量名、运算符和括号构成的字符串。数码照片存储成由0和1的序列构成的位串。事实上,所有由计算机存储和操作的数据都可以用位串来表示。作为计算机的用户,我们经常以文本字符串的形式向计算机输入信息以及接收计算机的输出。计算机程序的源代码是由关键字、标识符和特殊字符(这些构成了编程语言的字母表)组成的文本字符串。由于字符串十分重要,所以我们在本章中首先形式化地定义字符串的表示,并研究在字符串上的操作的性质。

我们感兴趣的语言并不是由任意字符串构成的,并非所有由英语字母构成的字符串都是句子,也并非源代码的所有字符串都是合法的计算机程序。语言是由满足某些特定需求以及用于定义语言语法的限制条件的字符串所组成的。在本章中,我们将使用递归定义和集合操作来强化语言中字符串的语法限制。

[41]

同时,我们也将介绍正则表达式定义的语言族。正则表达式描述了一种模式,与这个正则表达式相关联的语言由所有匹配这种模式的字符串构成。尽管我们通过集合论构造的方法来介绍正则表达式,但是我们仍然可以看出这些语言的出现,就像由正则文法生成的以及有限状态自动机接收的语言的出现那样自然。本章的结尾我们将给出正则表达式在搜索和模式匹配上的应用。

2.1 字符串和语言

描述语言,首先要给出字母表的标识和语言中出现的符号的集合。语言的元素是字母表符号构成的有限长度字符串。因此,研究语言就需要理解产生和处理字符串的操作。在本节中,我们将给出字母表上的字符串和基本字符串操作的精确定义。

对字母表的惟一要求就是它包括有限不可分割的对象。自然语言的字母表,像英语和法语,包含语言的单词和停顿标记。语言字母表中的符号可以看作是是不可分割的对象。单词 language 不可以划分成 lang 和 usage。单词 format 和单词 for 与 mat 没有关系,它们都是独立的。字母表的字符串就是单词和停顿符号构成的序列。你读到的句子就是这种字符串。计算机语言的字母表包含合法关键词、标识符和语言的符号。这个字母表上的字符串就是源代码的序列。

因为语言字母表中的元素都是不可分割的,所以我们统一用单个字母来标识它们。不带下角标的字母 a, b, c, d, e 用来表示字母表中的元素, Σ 用来表示字母表。字母表上的字符串用接近字母表尾部的字母表示。特别地, p, q, u, v, w, x, y, z 用来表示字符串。自然语言和计算机语言的表示允许例外。在这种情况下,字母表包含特殊语言的不可分割元素。

可以使用字母表中元素的序列来非形式化地定义字符串。为了确定字符串的性质,字母表的字符串集合是递归定义的。最基本的字符串是不包含任何元素的字符串。这个字符串叫做空串 (null string),用 λ 表示。定义中采用的操作符主要包括邻接字母表中的一个元素到已知字符串的右侧。

定义 2.1.1 已知 Σ 是字母表, Σ^* 是 Σ 上的字符串的集合,它可以递归定义如下:

[42]

i) 基础步骤: $\lambda \in \Sigma^*$ 。

ii) 递归步骤: 如果 $w \in \Sigma^*$ 且 $a \in \Sigma$, 那么 $wa \in \Sigma^*$ 。

iii) 封闭: $w \in \Sigma^*$ 仅当它可以由通过有限步构造而成。

对于所有的非空字母表 Σ , Σ^* 包含无穷多个元素。如果 $\Sigma = \{a\}$, 那么 Σ^* 包含字符串 $\lambda, a, aa, aaa, \dots$ 。字符串 w 的长度从直觉上讲是字符串中元素的数目, 而在形式上则是构造字符串的过程中应用归纳的步数, 表示成 $\text{length}(w)$ 。如果 Σ 包含 n 个元素, 那么 Σ^* 中就包含 n^k 个长度为 k 的字符串。

例 2.1.1 已知 $\Sigma = \{a, b, c\}$ 。 Σ^* 中的元素包括

长度 0: λ

长度 1: a, b, c

长度 2: $aa, ab, ac, ba, bb, bc, ca, cb, cc$

长度 3: $aaa, aab, aac, aba, abb, abc, aca, acb, acc$

$baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc$

$caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc$

□

根据我们的非形式化定义, 语言包含字母表上的字符串。例如, 英语包含有单词构成的字符串, 我们称之为句子。并不是所有的单词构成的字符串都是句子, 只有那些要素单词的顺序和类型满足了某个条件的才是。有了描述正确句子形式的规则、需求和限制, 就可以定义语言的语法了。这些观察的结果帮助我们给出了语言的形式化定义: 语言是包含字母表上所有可能的字符串的集合的子集。

定义 2.1.2 字母表 Σ 上的语言 (language) 是 Σ^* 的子集。

因为字符串是语言的元素, 所以必须检查其上的字符串和操作符的性质。串联, 就是把两个字符串联在一起, 这是字符串生成的基本操作。串联的形式化定义是以串联的第二个字符串的长度为对象进行归纳的。在这一点上, 把字母表上的单个成员连接到字符串的最右端的基本操作, 就是我们到目前为止惟一介绍过的字符串上的操作。因此, 任何新的操作都必须基于它来定义。

定义 2.1.3 已知 $u, v \in \Sigma^*$ 。 u 和 v 的串联 (concatenation), 记作 uv , 是 Σ^* 上的二元操作, 定义如下:

i) 基础步骤: 如果 $\text{length}(v) = 0$, 那么 $v = \lambda$ 和 $uv = u$ 。

ii) 递归步骤: 已知 v 是长度 $\text{length}(v) = n > 0$ 的字符串。那么对于任意长度为 $n-1$ 的字符串 w 都有 $v = wa$, 并且如果 $a \in \Sigma$, 那么 $uv = (uw)a$ 。

例 2.1.2 已知 $u = ab$, $v = ca$ 并且 $w = bb$ 。那么

$$uv = abca$$

$$vw = cabb$$

$$(uv)w = abcabb$$

$$u(vw) = abcabb.$$

□

u, v 和 w 的串联的结果与操作使用的顺序无关。从数学上讲, 这个性质叫做相关性 (associativity)。定理 2.1.4 证明了串联是相关二元操作。

定理 2.1.4 已知 $u, v, w \in \Sigma^*$, 那么 $(uv)w = u(vw)$ 。

证明: 对字符串 w 长度的归纳。我们使用和字符串的递归定义兼容的方式选择 w , 这种方式从已知字符串的最右侧进行构造。

基础步骤: $\text{length}(w) = 0$, 那么 $w = \lambda$, 并且根据串联的定义有 $(uv)w = uv$ 。另一方面, $u(vw) = u(v) = uv$ 。

归纳假设: 假设对于所有长度不超过 n 的字符串 w 都有 $(uv)w = u(vw)$ 。

归纳步骤: 我们需要证明对于所有长度为 $n+1$ 的字符串 w 都有 $(uv)w = u(vw)$ 。设 w 是这样的字符串。那么对于长度为 n 的字符串 x , $a \in \Sigma$, 都有 $w = xa$, 并且

$$\begin{aligned} (uv)w &= (uv)(xa) && (\text{替换}, w = xa) \\ &= ((uv)x)a && (\text{串联定义}) \\ &= (u(vx))a && (\text{归纳假设}) \\ &= u((vx)a) && (\text{串联定义}) \\ &= u(v(xa)) && (\text{串联定义}) \\ &= u(vw) && (\text{替换}, xa = w). \end{aligned}$$

■

因为相关性保证了无论什么样的操作顺序都能得到同样的结果,所以串联应用中的括号可以忽略。人们通常使用指数来简化字符串自身的串联。因此, uu 可以写成 u^2 , uuu 写成 u^3 ,等等。为了保持完整性, u^0 表示字符串 u 串联自身0次,即定义为空字符串。串联的操作是不可交换的。对于字符串 $u=ab$, $v=ba$, $uv=abba$,而 $vu=baab$ 。注意 $u^2=abab$,而不是 $aabb=a^2b^2$ 。

子字符串可以使用串联来定义。从直观上来说,如果 u “发生在 v 之内”,那么 u 就是 v 的子字符串。形式上,如果存在字符串 x 和 y 使得 $v=xuy$,那么 u 就是 v 的子字符串。当 x 是空串,那么 v 的子字符串 u 就是 v 的前缀(prefix),即 $v=uv$ 。类似地,如果 $v=xu$,那么 u 就是 v 的后缀(suffix)。

字符串的逆就是把字符串从后往前写。 $Abbc$ 的逆是 $cbba$ 。与串联类似,这个一元操作也是通过字符串的长度递归定义的。从一个字符串的最右侧去掉一个元素形成一个比较小的字符串,可以把这种方法用在递归定义中。定理2.1.6构造了串联和逆操作的关系。

定义 2.1.5 已知 u 是 Σ^* 中的一个字符串, u 的逆(reversal),记成,定义如下:

- i) 基础步骤:如果 $\text{length}(u)=0$,那么 $u=\lambda$,并且 $\lambda^R=\lambda$ 。
- ii) 递归步骤:如果 $\text{length}(u)=n>0$,那么对于某个长度为 $n-1$ 的字符串 w 和某个 $a\in\Sigma$ 有 $u=wa$ 并且 $u^R=aw^R$ 。

定理 2.1.6 已知 $u, v\in\Sigma^*$,那么 $(uv)^R=v^Ru^R$ 。

证明:对字符串 v 的长度进行归纳证明。

基础步骤:如果 $\text{length}(v)=0$,那么 $v=\lambda$,并且 $(uv)^R=u^R$ 。类似地, $v^Ru^R=\lambda^Ru^R=u^R$ 。

归纳假设:假设对于所有长度不超过 n 的字符串 v , $(uv)^R=v^Ru^R$ 。

归纳步骤:我们必须证明,对于任何长度为 $n+1$ 的字符串 v 都有 $(uv)^R=v^Ru^R$ 。设 v 是长度为 $n+1$ 的字符串。那么 $v=wa$,其中 w 是长度为 n 的字符串,并且 $a\in\Sigma$ 。按照归纳步骤构造得,

$$\begin{aligned}
 (uv)^R &= (u(wa))^R \\
 &= ((uw)a)^R && \text{(串联的相关性)} \\
 &= a(uw)^R && \text{(逆的定义)} \\
 &= a(w^Ru^R) && \text{(归纳假设)} \\
 &= (aw^R)u^R && \text{(串联的相关性)} \\
 &= (wa)^Ru^R && \text{(逆的定义)} \\
 &= v^Ru^R.
 \end{aligned}$$

2.2 语言的有穷规格说明

语言是由字母表上的字符串构成的集合。我们感兴趣的语言不是字符串的任意集合,而是那些满足某种语法要求的字符串。语言的规约要求对于语言的字符串的描述没有二义性。有限语言可以通过显式地列举它的元素来定义。满足简单语法要求的几种简单无穷语言都是按照下面的方式递归定义的。

例 2.2.1 字母表 $\{a,b\}$ 上的字符串构成的语言 L ,其中每个字符串均以 a 为首且长度为偶数,其定义如下:

- i) 基础步骤: $aa, ab\in L$ 。
- ii) 递归步骤:如果 $u\in L$,那么 $uaa, uab, uba, ubb\in L$ 。
- iii) 封闭:字符串 $u\in L$,仅当它可以由基础元素经过有限递归步骤构造而成。

L 中的字符串是通过把两个元素添加到以前构造的字符串的最右端构造而成的。基础保证 L 中的每个字符串都是以 a 为首字母的。增加长度为2的子字符串保证了字符串的长度是偶数。□

例 2.2.2 字母表 $\{a,b\}$ 上的语言 L 定义为

- i) 基础步骤: $\lambda\in L$;
- ii) 递归步骤:如果 $u\in L$,那么 $ua, uab\in L$ 。
- iii) 封闭:字符串 $u\in L$,仅当它可以由基础元素经过有限步骤构造而成。

这个语言包含了 a 后面紧跟着 b 的所有字符串。例如, $\lambda, a, abaab$ 属于 L ,而 bb, bab, abb 不属于 L 。□

前面例子中的归纳步骤把元素串联到已知字符串的一端。而把一个字符串拆分成子字符串,就是在原字符串中的任何位置增加元素。这种技术可以用下面的例子来解释。

例 2.2.3 已知 L 是字母表 $\{a, b\}$ 上的语言, 其定义为

i) 基础步骤: $\lambda \in L$;

ii) 递归步骤: 如果 $u \in L$ 和 u 可以写成 $u = xyz$, 那么 $xaybz \in L$ 。

iii) 封闭: 字符串 $u \in L$, 仅当它可以由基础元素使用有限步骤构造而成。

语言 L 包括所有具有相同数目的 a 和 b 的字符串。在归纳步骤的第一次构造过程中, 即 $xaybz \in L$, 包括下面三个操作:

- 选择已经属于 L 的字符串 u 。
- 把 u 划分成子字符串 x, y, z , 使得 $u = xyz$ 。而且, 这几个子字符串也可以是 λ 。
- 把 a 插入到 x 和 y 中间, 而把 b 插入到 y 和 z 中间。

采取了这些步之后, 这两个规则就可以直观地解释为“把一个 a 和一个 b 插入到字符串 u 中的任何位置”。

递归定义提供了一种定义语言字符串的工具。例 2.2.1、例 2.2.2 和例 2.2.3 表明序列、位置和奇偶这几方面的要求可以通过字符串的递归方法来生成。然而, 这种方法却不适用于计算语言或自然语言的复杂语法要求。

另一种构造语言的技术就是使用集合操作, 由简单的字符串集合构造字符串的复杂集合。定义在字符串上的操作可以被扩充到集合上, 甚至是语言上。无穷语言的描述可以使用集合操作由有限集合来构造。下面两个定义介绍了字符串集合上的操作, 它可以用于语言定义和模式规约中。

定义 2.2.1 语言 X 和 Y 的串联, 记作 XY , 表示语言

$$XY = \{uv \mid u \in X \text{ 并且 } v \in Y\}.$$

X 与自身串联 n 次记成 X^n 。 X^0 可以定义为 $\{\lambda\}$ 。

例 2.2.4 已知 $X = \{a, b, c\}$ 和 $Y = \{abb, ba\}$ 。那么

$$XY = \{aabb, babb, cabb, aba, bba, cba\}$$

$$X^0 = \lambda$$

$$X^1 = X = \{a, b, c\}$$

$$X^2 = XX = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

$$X^3 = X^2X = \{aaa, aab, aac, aba, abb, abc, aca, acb, acc, \\ baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, \\ caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}.$$

前面例子中的集合看起来很相似。对于每个 i , X^i 包含例 2.1.1 中给出的 Σ^* 中长度为 i 的字符串。这些观察结果形成了另一个集合操作, 即集合 X 的克林 (Kleene) 星, 记作 X^* 。使用 $*$ 操作, 集合上的字符串就可以使用串联和并, 而不是使用定义 2.1.1 中的基本操作来定义了。

定义 2.2.2 已知 X 是集合。那么

$$X^* = \bigcup_{i=0}^{\infty} X^i \text{ 并且 } X^+ = \bigcup_{i=1}^{\infty} X^i$$

集合 X^* 包含由 X 的元素构造的所有字符串。如果 X 是一个字母表, 那么 X^* 就是 X 上所有非空字符串的集合。 X^+ 的另一个定义是使用串联和克林星来定义, 即 $X^+ = XX^*$ 。

形式语言的定义需要对属于该语言的字符串给出非二义性的说明。非形式化地定义语言会缺少准确定义的严格性。考察 a, b 上所有包含子串 bb 的字符串的语言。这是不是意味着这种语言的字符串包含那些只出现过一次子串 bb 的字符串, 或者由多个子字符串 bb 构成的字符串是否被这种语言所允许? 这些问题可以通过明确描述包含一次出现 bb , 或者至少出现一次的字符串来回答。然而, 这些问题总是必然存在于自然语言提供的不准确方法中的。

集合操作的准确性可以用于对语言字符串的非二义性描述。例 2.2.5 给出了包含子串 bb 的字符串的理论定义。根据这个定义, 我们可以知道这种语言包含 bb 至少出现一次的字符串。

例 2.2.5 语言 $L = \{a, b\}^* \{bb\} \{a, b\}^*$ 包含 $\{a, b\}^*$ 中所有包含子串 bb 的字符串, $\{bb\}$ 的串联, 其中包含单串 bb , 保证 L 中每个字符串都包含 bb 。集合 $\{a, b\}^*$ 允许在 bb 的前面或后面出现任意数目的 a 和 b , 它们可以按照任何顺序出现。特别地, 子串 bb 的另一个复制也可以出在串联 $\{bb\}$ 的前面或后面。□

例 2.2.6 串联可以用来描述字符串的构成成分之间的顺序。已知 L 是包含所有串首为 aa 、串尾为 bb 的字符串。集合 $\{aa\} \{a, b\}^*$ 描述了前缀为 aa 的字符串。类似地, $\{a, b\}^* \{bb\}$ 是所有后缀为 bb 的字符串的集合。因此 $L = \{aa\} \{a, b\}^* \cup \{a, b\}^* \{bb\}$ 。□

例 2.2.7 已知 $L_1 = \{bb\}$, $L_2 = \{\lambda, bb, bbbb\}$ 是属于 $\{b\}$ 上的语言。语言 L_1^* 和 L_2^* 都包含那些由偶数个 b 构成的字符串。注意, 长度为 0 的 λ 也是属于 L_1^* 和 L_2^* 的元素。□

例 2.2.8 集合 $\{aa, bb, ab, ba\}^*$ 包含所有 $\{a, b\}$ 上偶数长度的字符串。不断使用串联, 每次增加两个元素来构造新的字符串。奇数长度的字符串的集合可以定义为 $\{a, b\}^* - \{aa, bb, ab, ba\}^*$ 。这个集合可以通过把一个元素串联到一个偶数长度的字符串来获得。因此, 奇数长度的字符串也可以通过 $\{aa, bb, ab, ba\}^* \{a, b\}$ 来定义。□

2.3 正则集合和表达式

在前面的小节中, 我们根据已有的语言使用集合操作来构造新语言。这些操作用来保证某种模式出现在语言的字符串中。在本节中, 我们将使用构造语言的集合操作方法, 但是会限制那些构造过程中使用的集合和操作。

空集, 由空串构成的集合, 以及使用并、串联和克林星操作构成的字符串的集合都是正则的 (regular)。定义 2.3.1 中递归定义的正则集合, 包含一族语言。这族语言在形式语言、模式识别和有限状态机理论中都扮演着重要的角色。

定义 2.3.1 已知字母表 Σ , 那么 Σ 上的正则集合 (fuzzy sets) 递归定义为

i) 基础步骤: 对于任意的 $a \in \Sigma$, \emptyset , $\{\lambda\}$ 和 $\{a\}$ 都是 Σ 上的正则集合。

ii) 递归步骤: 已知 X 和 Y 都是 Σ 上的正则集合。那么集合

$$\begin{aligned} X \cup Y \\ XY \\ X^* \end{aligned}$$

都是 Σ 上的正则集合。

iii) 封闭: X 是 Σ 上的正则集合, 仅当它可以由基础元素由有限步递归定义生成。

如果一种语言可以用一个正则集合来定义, 那么这种语言就是正则的 (regular)。下面的例子给出了如何用正则集合来描述语言的字符串。

例 2.3.1 例 2.2.5 中的语言, 即包含子串 bb 的字符串的集合, 是这个集合是 $\{a, b\}$ 上的正则集合。根据定义的基础, $\{a\}$ 和 $\{b\}$ 都是正则集合。 $\{a\}$ 和 $\{b\}$ 的并以及克林星操作的结果都是 $\{a, b\}^*$, 这也就是 $\{a, b\}$ 上的所有字符串构成的集合。使用串联后, $\{b\} \cdot \{b\} = \{bb\}$ 就是正则的。两次应用串联就可以得到 $\{a, b\}^* \{bb\} \{a, b\}^*$ 。□

例 2.3.2 以 a 为首字母和尾字母, 并且至少有一个 b 的字符串的集合, 是 $\{a, b\}$ 上的正则集合, 这个集合上的字符串可以直观地描述为“以 a 为首, 跟着任意字符串, 然后是 b , 接着又是任意字符串, 最后是 a 。”串联

$$\{a\} \{a, b\}^* \{b\} \{a, b\}^* \{a\}$$

展示了集合的正则性。□

根据定义, 正则集合是那些可以通过空集构造的集合, 包含空串的集合, 以及对字母表上的单一元素进行并、串联和克林星操作而获得的集合。正则表达式可以用来简化正则集合的表示。正则集合 \emptyset , $\{\lambda\}$ 和 $\{a\}$ 使用 \emptyset , λ 和 a 来表示, 而不需要集合括号。集合操作并、克林星和串联分别使用 \cup , * 和并列来表示。圆括号用来表示操作的顺序。

定义 2.3.2 已知字母表。上的正则表达式 (regular expressions) 递归定义如下:

- i) 基础步骤: 对于任意的 $a \in \Sigma$, \emptyset , λ 和 a 都是 Σ 上的正则表达式。
 ii) 递归步骤: 已知 u 和 v 都是 Σ 上的正则表达式。那么表达式

$$(u \cup v)$$

$$(uv)$$

$$(u^*)$$

都是 Σ 上的正则表达式。

- iii) 封闭: u 是 Σ 上的正则表达式, 仅当它可以由基础元素由有限步递归定义生成。

因为并和串联是类似的, 所以在表示一系列这些操作的序列的时候, 圆括号是可以省略的。为了进一步减少圆括号的数目, 这些操作被赋予了一定的优先级。克林星的优先级最高, 紧接着是并和串联。应用这些规则, 例 2.3.1 和例 2.3.2 中的集合对应的正则表达式是 $(a \cup b)^* bb(a \cup b)^*$ 和 $a(a \cup b)^* b(a \cup b)^* a$ 。 u^* 表示表达式 uu^* 的缩写。类似的, u^+ 表示正则表达式 uu, u^+u , 等等。

例 2.3.3 定义在 a, b 上的集合 $\{bawab \mid w \in \{a, b\}^*\}$ 是正则的。下面的表格给出了正则集合的递归生成步骤, 以及相应语言的正则表达式定义。表中右边一列给出了在递归操作中使用的每个构成成分是正则的理由。

集 合	表 达 式	理 由
1. $\{a\}$	a	基础
2. $\{b\}$	b	基础
3. $\{a\} \cup \{b\} = \{ab\}$	ab	1, 2, 串联
4. $\{a\} \cup \{b\} = \{a, b\}$	$a \cup b$	1, 2, 并
5. $\{b\} \cup \{a\} = \{ba\}$	ba	2, 1, 串联
6. $\{a, b\}^*$	$(a \cup b)^*$	4, Kleene 星
7. $\{ba\} \cup \{a, b\}^*$	$ba(a \cup b)^*$	5, 6, 串联
8. $\{ba\} \cup \{a, b\}^* \cup \{ab\}$	$ba(a \cup b)^* ab$	7, 3, 串联

前面的这个例子揭示了正则集合和正则表达式是如何从基本正则集合演化而来的。每个正则集合都可以通过例 2.3.3 给出的类似的操作的有限步分解来获得。

正则表达式定义了一种模式, 于是, 仅当字符串匹配该模式时, 这个字符串才属于这个正则表达式表示的语言。串联指定了顺序, 字符串 w 属于 uv 仅当它包含 u , 并且 u 后面紧跟着的就是 v 的字符串。Kleene 星允许重复和 \cup 选择。例 2.3.3 中的正则表达式给出的模式, 需要字符串的首是 ba , 而尾是 ab , 在这个前缀和后缀间的是 a 和 b 的任意组合。接下来的例子进一步解释了正则表达式描述模式的能力。

例 2.3.4 正则表达式 $(a \cup b)^* aa(a \cup b)^*$ 和 $(a \cup b)^* bb(a \cup b)^*$ 分别表示包含 aa 和 bb 的字符串。使用 \cup 操作把这两个表达式结合起来, 会得到表达式 $(a \cup b)^* aa(a \cup b)^* \cup (a \cup b)^* bb(a \cup b)^*$, 这个表达式表示的是 $\{a, b\}$ 上包含子串 aa 或 bb 的所有字符串的集合。 \square

例 2.3.5 $\{a, b\}$ 上只包含两个 b 的字符串的集合对应的正则表达式, 必须保证两个 b 的存在。 b 的前面、中间和后面可以包含任意数目的 a 。连接那些满足要求的子表达式可以得到 $a^* ba^* ba^*$ 。 \square

例 2.3.6 正则表达式

i) $a^* ba^* b(a \cup b)^*$

ii) $(a \cup b)^* ba^* ba^*$

iii) $(a \cup b)^* b(a \cup b)^* b(a \cup b)^*$

定义了 $\{a, b\}$ 上包含至少两个 b 的字符串的集合。正如例 2.3.5 所示, 至少两个 b 的存在是通过串联两个表达式 b 来保证的。 \square

例 2.3.7 考查通过表达式 $a^* (a^* ba^* ba^*)^*$ 定义的正则集合。表达式中圆括号里面的就是例 2.3.5 中只含有两个 b 的正则表达式。克林星构造了任意数目的这样的字符串的串联。结果是空串 (没有重复任何模式) 和包含正偶数 b 的所有字符串。只包含一个 a 的字符串不属于 $(a^* ba^* ba^*)^*$ 。

在表达式的开始串联 a^* , 就生成了具有偶数 b 的字符串。这个集合的另一个正则表达式是 $a^*(ba^*ba^*)^*$ 。□

例 2.3.8 使用可共享的子串使得正则表达式——首字母为 ba , 尾字母是 ab , 并且包含子串 aa ——的构造更加复杂了。表达式 $ba(a \cup b)^*aa(a \cup b)^*ab$ 中明确插入了这三个构成成分。每个字符串都必须包含至少四个 a 。但是, 字符串 $baab$ 满足上述条件, 可是它只有两个 a 。这种语言的正则表达式是

$$\begin{aligned} & ba(a \cup b)^*aa(a \cup b)^*ab \\ & \cup baa(a \cup b)^*ab \\ & \cup ba(a \cup b)^*aab \\ & \cup baab. \end{aligned}$$

□

正则表达式的构造过程是正向的, 使用串联、并, 或者 Kleene 星操作可以在表达式中明确地插入一些具有期望特点的表达。这里不存在逆向操作, 比如省略那些具有特殊性质的字符串。为了由没有性质的字符串集合来构造正则表达式, 必须首先正向地表示语言的构造条件, 然后再形式化地构造正则表达式。下面这两个例子就是解释这个方法的。

例 2.3.9 为了给 a, b 上不以 aaa 结尾的字符串集合构造正则表达式, 我们必须保证 aaa 不是所描述的表达式的任意字符串后缀。带有 b 的字符串的可能结尾包括 b , ba 或者 baa 。正则表达式的第一部分

$$(a \cup b)^*(b \cup ba \cup baa) \cup \lambda \cup a \cup aa$$

就定义了这些字符串。最后的三个表达式表示了长度为 0、1 和不包含 b 的长度为 2 的字符串的特殊例子。□

例 2.3.10 使用 $c^*(b \cup ac^*)^*$ 定义的语言 L 包含 a, b, c 上的所有不包含子串 bc 的字符串。外部的 c^* 和括号里的 ac^* 使得任意数目的 a 和 c 都可以按照任何顺序出现。 b 后面可以紧跟着一个 b , 或是 ac^* 。在 ac^* 首部的 a 直接阻挡了接下来的 c 。为了更好地理解表达式表示的集合, 我们要试图说明 $acabacc$ 和 $bbaaacc$ 属于 $c^*(b \cup ac^*)^*$ 表示的集合。□

例 2.3.6 和例 2.3.7 证明了语言的正则表达式的定义不是惟一的。表示同一集合的两个表达式是等价的 (equivalent)。代数演化正则表达式使用表 2-1 中的实体来构造等价关系。这些实体是正则表达式的并、串联和 Kleene 星操作的性质。

表 2-1 正则表达式恒等式

1. $\emptyset u = u \emptyset = \emptyset$	9. $u(v \cup w) = uv \cup uw$
2. $\lambda u = u \lambda = u$	10. $(u \cup v)w = uw \cup vw$
3. $\emptyset^* = \lambda$	11. $(uv)^*u = u(vu)^*$
4. $\lambda^* = \lambda$	12. $(u \cup v)^* = (u^* \cup v^*)^*$
5. $u \cup v = v \cup u$	$= u^*(u \cup v)^* = (u \cup vu^*)^*$
6. $u \cup \emptyset = u$	$= (u^*v^*)^* = u^*(vu^*)^*$
7. $u \cup u = u$	$= (u^*v)^*u^*$
8. $u^* = (u^*)^*$	

等式 5 体现了集合并的交换性。等式 9 和等式 10 是把并和串联的分配律用正则表达式来表示。表达式的最终集合提供了由元素 u 和 v 构造的所有字符串的一定数量的等价表示。表 2-1 可以用来建立或简化正则表达式的等价关系。

例 2.3.11 构造 a, b 上不包含子串 aa 的字符串的集合所对应的正则表达式。这个集合中的字符串可能包含任意数目的 b 作为前缀。所有的 a 后面都紧跟着至少一个 b , 或者字符串在此处结束。正则表达式 $b^*(ab^*)^* \cup b^*(ab^*)^*a$ 生成的集合可以分割成两个不相交的子集合。第一部分包括以 b 结

尾的字符串，第二部分包括以 a 结尾的字符串。使用表 2-1 中的恒等式化简这个表达式可以得到

$$\begin{aligned} & b^*(ab^+)^* \cup b^*(ab^+)^*a \\ &= b^*(ab^+)^*(\lambda \cup a) \\ &= b^*(abb^+)^*(\lambda \cup a) \\ &= (b \cup ab)^*(\lambda \cup a). \end{aligned}$$

□

正则表达式可以用来描述很多复杂的模式，但是值得注意的是，仍然有很多语言是无法用任何正则表达式来定义的。在第 6 章，我们将看到正则表达式无法定义语言 $\{a^ib^i \mid i \geq 0\}$ 。

2.4 正则表达式和文本搜索

对于大多数的计算机用户而言，正则表达式的一个常见应用是搜索文档和文件。在本节中，我们将给出正则表达式在文本搜索中的两类应用。

使用正则表达式定义语言与用其进行文本搜索的主要区别就是期望匹配的区域。如果整个字符串与正则表达式所描述的模式相匹配，那么这个字符串就属于该正则表达式定义的语言。例如，字符串匹配 ab^+ 仅当它开头是 a ，紧跟着一个或多个 b 。

在文本搜索中，我们在文本中寻找符合预期模式的子串。因此，单词

about
abbot
rehabilitate
tabulate
abominable

可以看成是匹配模式 ab^+ 的。事实上，*abominable* 两次匹配这个模式。

于是，就产生了两种不同类型的文本匹配，它们分别称为离线搜索和在线搜索。离线搜索指的是，一个搜索程序在运行，程序的输入是模式和文件，输出包括匹配模式的行或文档。离线搜索常常是使用操作系统设备或者使用专门为搜索设计的语言编写的程序。GREP 和 awk 就是文档搜索的例子，而 Perl 则是用于文件搜索的编程语言。我们将使用 GREP (Global search for Regular Expression and Print) 的首字母缩写——来解释这种类型的正则表达式搜索。

文本浏览器、文本编辑器和文字处理系统都提供了在线搜索工具。在线搜索的目的是为了交互式地发现第一个、下一个，进而发现所有的匹配搜索模式的子串。微软 Word 中的“发现”命令可以用于展示在线和离线模式匹配的区别。

因为期望模式通常是使用键盘输入的，所以搜索设备使用的正则表达式表示通常都很简练，而且不包含上角标。尽管在搜索应用中使用正则表达式没有固定的语法，但是大部分应用使用的表示还是有很多共性的。我们将使用 GREP 的扩展正则表达式表示法来解释文本搜索中的模式描述。

文件或文档的字母表经常包括 ASCII 字符表，参见附录 III。它所包含的元素数量是我们在正则表达式中使用的字母表的两倍或三倍。使用字母表 a, b ，任何字符串的正则表达式都属于 $(a \cup b)^*$ 。使用这种形式的任何 ASCII 字符串的表达式都需要写几行，因此使用键盘输入十分不方便。我们将介绍两种表示习惯——括号表达式和范围表达式，从而简化模式匹配在扩展字母表上的描述。

括号表示 $|$ 用来表示字母表字符的并。例如， $[abcd]$ 用来表示表达式 $(a \cup b \cup c \cup d)^*$ 。在左边括号的后面加一个 c 符号，就产生了并的补。因此， $[^abcd]$ 表示所有除 a, b, c 和 d 以外的字符。

范围表达式使用 ASCII 码序列来描述字符的序列。例如， $A-Z$ 是表示所有大写字母的范围表达式。在 ASCII 表中这些是编码从 65 到 90 的字符。范围表达式可以使用括号表达式作为参数， $[a-zA-Z0-9]$ 表示所有字母和数字的集合。另外，某种经常出现的字符的子集可以使用它们的记忆单元标识符。例如， $[:digit:]$ ， $[:alpha:]$ 和 $[:alnum:]$ 是 $[0-9]$ ， $[a-zA-Z]$ 以及 $[a-zA-Z0-9]$ 的缩写。扩展的正则表达式表示也包含了用于表示匹配的单词头和尾的符号 \backslash 和 \wedge 。

除了标准操作 \cup 、串联和 $*$ 之外，GREP 的扩展正则表达式表示还包含表达式上的其他操作。这些操作虽然没有扩展表述的模式类型，但是它们简化了模式的表示。表 2-2 给出了扩展表达式操作的

[54]

[55]

表示。优先级和括号进一步定义了操作的范围。

表 2-2 扩展的正则表达式操作

操作	符号	例子	正则表达式
连接		ab	<i>ab</i>
		[a-c][AB]	<i>aA ∪ aB ∪ bA ∪ bB ∪ cA ∪ cB</i>
Kleene 星	*	[ab]*	<i>(a ∪ b)*</i>
分离		[ab]* A	<i>(a ∪ b)* ∪ A</i>
0 或多个	+	[ab] +	<i>(a ∪ b)+</i>
0 或 1 个	?	a?	<i>(a ∪ λ)</i>
一个字符	.	a.a	<i>a(a ∪ b)a</i> if $\Sigma = \{a, b\}$
n 次	{n}	a 4	<i>aaaa = a⁴</i>
n 或多次	{n,}	a 4,	<i>aaaaa*</i>
n 到 m 次	{n,m}	a 4,6	<i>aaaa ∪ aaaaa ∪ aaaaaa</i>

GREP 的输入是模式和待搜索的文件。GREP 在文件中进行逐行搜索。如果某行包括匹配模式的子串,那么这行就被打印出来,然后继续对下面的行进行搜索。为了使用扩展的正则表达式进行模式匹配,我们将在莎士比亚的《凯撒大帝》第二场第二幕中搜索凯撒对他妻子评价

Cowards die many times before their deaths;
The valiant never taste of death but once.
Of all the wonders that I yet have heard,
It seems to me most strange that men should fear;
Seeing that death, a necessary end,
Will come when it will come.

[56]

我们首先寻找匹配模式 $m[a-z]n$ 的。这就是寻找长度为 3, 包含 m 和 n , 并且它们中间是任何一个小写字母的字符串。搜索的结果是

```
C: >grep-E "m[a-z]n" caesar
Cowards die many times before their deaths;
It seems to me most strange that men should fear;
```

GREP 中的 -E 选项是描述模式的扩展正则表达式表示法,而引号则描绘了模式 many 中的子串 man 和单词 men 匹配了这个模式,因此包含这些字符串的行被打印了出来。

现在搜索变成了寻找 m 和 n 之间被任何小写字母或空字符分割的出现。

```
C: >grep-E "m[a-z]*n" caesar
Cowards die many times before their deaths;
It seems to me most strange that men should fear;
Will come when it will come.
```

输出部分增加了最后一行,这是因为这行的 *me* 和 *when* 与模式相匹配。模式 $m[a-z]*n$ 在下面这行得到了六次匹配:

```
It seems to me most strange that men should fear;
```

然而, GREP 不需要找到所有的匹配,每行中找到一个就可以把对应的行输出

扩展正则表达式表示法可以用来描绘文本中更加复杂的模式。考虑在文本文件中寻找包含人名的行。为了确定名字的形式,我们首先考虑所有作为名字部分组成的可能字符串

- i) 第一个名字或者初始名: $[A-Z][a-z]^+ | [A-Z][.].$
- ii) 中间名字, 初始名, 或者两者都不是: $([A-Z][a-z]^+ | [A-Z][.].)?$
- iii) 姓: $[A-Z][a-z]^+$

出现在第一个位置上的字符串或者是名字或者是初始名字。如果前者的话,那么字符串的首字母应该是大写的,紧接着的是由小写的字母构成的字符串。初始名字仅仅就是一个大写字母,然后紧跟

着一个句点。同样的表达式可以用于中间名字或姓。问号“?”表示没有中间名字,或者是需要初始名字。这些表达式都可以用下面显示的空格来连接。

$([A-Z][a-z] + |[A-Z][.])((([A-Z][a-z] + |[A-Z][.])[])?([A-Z][a-z] +)$

57 就是一个通用的匹配名字的模式。

接下来的表达式将匹配 E. B. White, Edgar Allen Poe 和 Alan Turing。因为模式匹配是严格限制于字符串形式的,并没有潜在意义的(即:模式匹配检查语法,但不检查语义),因此表达式也匹配 Buckingham Palace 和 U. S. Mail。而且,模式不匹配 Vincent van Gogh, Dr. Watson 或者 Aristotle。为了匹配名字的这些变化,还需要增加其他的条件。

与离线分析不同,网页浏览和文字处理中的搜索命令,是以交互式的方式来发现匹配输入模式的字符串的。子串匹配一个模式可以分布到很多行。使用微软 Word 中的“寻找”命令来匹配模式 $m * n$,就是寻找首字母是 m ,尾字母是 n 的子串,在 m 和 n 之间可以是任何字符串。搜索的结果将是发现并强调匹配该模式的第一个子串的之前的位置或是之后的位置。重复点击“下一个”将会继续搜索,并会突出模式的下一个匹配。在凯撒中匹配 $m * n$ 的子串如下,其中,匹配的子串会突出显示。

Cowards die *many* times before their deaths;
 Cowards die many times *before* their deaths;
 The valiant *never* taste of death but once.
 It seems to me *most strange* that men should fear;
 It seems to me *most strange* that men should fear;
 It seems to me *most strange* that men should fear;
 It seems to me most strange that *men* should fear;
 Will come when it will come.

请注意,并不是所有匹配的子串都会被突出显示。模式 $m * n$ 匹配所有满足下面条件的字符串:首字母是 m ,并且子串后面出现 n 。这个搜索只强调文件中每个 m 的第一次匹配出现。

在第 6 章中我们将看到正则表达式可以转化成有限状态机。通过这个机器的计算过程,就可以找到匹配该表达式描述的模式的字符串或子串。使用正则表达式的操作限制——交和集合差是不允许的——使得把模式的描述转化成搜索算法的实现的过程更容易了。

2.5 练习

58

- 给出 Σ 上的字符串的长度的递归定义。使用字符串定义中的主要操作。
- 对 i 进行递归,证明对于任意 w 和所有 $i \geq 0$, 都有 $(w^R)^i = (w^i)^R$ 。
- 对字符串的长度进行递归,证明对于所有的字符串 $w \in \Sigma^*$, 都有 $(w^R)^R = w$ 。
- 已知 $X = \{aa, bb\}$ 和 $Y = \{\lambda, b, ab\}$ 。
 - 列出集合 XY 中的字符串。
 - X^* 中有多少个字符串的长度是 6?
 - 列出集合 Y^* 中长度不超过 3 的字符串。
 - 列出 X^*Y^* 中长度不超过 4 的字符串。
- 已知 L 是 $\{a, b\}$ 上通过归纳定义的字符串的集合。
 - 基础步骤: $b \in L$ 。
 - 递归步骤: 如果 u 属于 L , 那么 $ub \in L, uab \in L, uba \in L$ 并且 $bua \in L$ 。
 - 封闭: 字符串 v 属于 L 仅当它可以由基础通过有限步归纳而成。
 - 列出集合 L_0, L_1 和 L_2 的元素。
 - 字符串 $bbaaba$ 属于 L 吗? 如果是, 它是如何产生的。如果不是, 解释原因。
 - 字符串 $bbaaaabb$ 属于 L 吗? 如果是, 它是如何产生的。如果不是, 解释原因。
- 给出 $\{a, b\}$ 上包含至少一个 b , 并且在第一个 b 之前有偶数个 a 的字符串的集合的定义。例如, $bab, aabb$ 和 $aaaabababab$ 属于这个集合, 可是 aa, abb 就不是。

7. 给出集合 $\{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$ 的递归定义。
8. 给出 $\{a, b\}$ 上 a 的个数是 b 的两倍的字符串的集合的递归定义。
9. 证明: 例 2.2.1 中定义的语言的每个字符串的长度都是偶数。可以依据对字符串的递归构造过程来进行归纳证明。
10. 证明: 例 2.2.2 中定义的语言的每个字符串中 a 的个数都不少于 b 。已知 $n_a(u)$ 表示字符串中 a 的个数, $n_b(u)$ 表示 u 中 b 的个数。归纳证明应该建立不等关系 $n_a(u) \geq n_b(u)$ 。
11. 已知 L 是 $\{a, b\}$ 上递归定义的:
 - i) 基础步骤: $\lambda \in L$ 。
 - ii) 递归步骤: 如果 $u \in L$, 那么 $aaub \in L$ 。
 - iii) 封闭: 字符串 w 属于 L , 仅当它可以由基础使用有限步递归步骤构造而成。
 - a) 给出使用递归定义的集合 L_0, L_1 和 L_2 。
 - b) 给出使用递归定义的字符串集合的固有定义。
 - c) 使用数学推导证明每个字符串 u 都属于 L , 要求字符串中 a 的个数是 b 的个数的二倍。已知 $n_a(u)$ 和 $n_b(u)$ 表示 u 中 a 的个数和 b 的个数。
12. 字母表 Σ 上的回文 (palindrome) 指的是: Σ^* 从前往后与从后向前拼写完全相同的字符串。 Σ 上的回文构成的集合, 可以通过下面的递归来定义:
 - i) 基础步骤: 对于所有的 $a \in \Sigma$, λ 和 a 都是回文。
 - ii) 递归步骤: 如果 w 是回文, 并且 $a \in \Sigma$, 那么 awa 是回文。
 - iii) 封闭: w 是回文, 仅当它可以由基础元素使用有限步递归而获得。
 回文的集合可以定义为 $\{w \mid w = w^R\}$ 。证明这两个定义产生的是同一个集合。
13. 已知 $L_1 = \{aaa\}^*$, $L_2 = \{a, b\} \{a, b\} \{a, b\} \{a, b\}$ 以及 $L_3 = L_2^*$ 。描述语言 L_2, L_3 和 $L_1 \cap L_3$ 中的字符串。
从练习 14 到 38, 给出描述下面集合的正则表达式。
 14. $\{a, b, c\}$ 上的字符串的集合, 要求所有的 a 在 b 的前面, b 在 c 的前面。当然也有可能没有 a , 没有 b 或者没有 c 。
 15. 练习 14 中的集合, 但是不包括空串。
 16. $\{a, b, c\}$ 上长度为 3 的字符串的集合。
 17. $\{a, b, c\}$ 上长度小于 3 的字符串的集合。
 18. $\{a, b, c\}$ 上长度大于 3 的字符串的集合。
 19. $\{a, b\}$ 上包含子串 ab , 并且长度大于 2 的字符串的集合。
 20. $\{a, b\}$ 上长度不小于 2, 并且所有的 a 都在 b 的前面的字符串的集合。
 21. $\{a, b\}$ 上包含子串 aa 和 bb 的字符串的集合。
 22. $\{a, b\}$ 上子串 aa 出现至少两次的字符串的集合。提示: 注意子串 aaa 。
 23. $\{a, b, c\}$ 上开始字母为 a , 准确包含两个 b , 并且以 cc 结尾的字符串的集合。
 24. $\{a, b\}$ 上包含子串 ab 和子串 ba 的字符串的集合。
 25. $\{a, b, c\}$ 上每个 b 后面紧跟着至少一个 c 的字符串的集合。
 26. $\{a, b\}$ 上 a 的数目可以被 3 整除的字符串的集合。
 27. $\{a, b, c\}$ 上 b 和 c 的总数是 3 的字符串的集合。
 28. $\{a, b\}$ 上每个 a 的前面就是 b , 后者后面紧跟着 b 的字符串的集合。比如, $baab, aba$ 和 b 。
 29. $\{a, b, c\}$ 上不包含子串 aa 的字符串的集合。
 30. $\{a, b\}$ 上不是以 aaa 为首的字符串的集合。
 31. $\{a, b\}$ 上不包含子串 aaa 的字符串的集合。
 32. $\{a, b\}$ 上不包含子串 aba 的字符串的集合。
 33. $\{a, b\}$ 上子串 aa 只出现一次的字符串的集合。
 34. $\{a, b\}$ 上长度为奇数, 并且包含字符串 bb 的字符串的集合。

35. $\{a, b, c\}$ 上长度为偶数, 并且一定包含一个 a 的字符串的集合。
36. $\{a, b\}$ 上长度为奇数, 并且一定只包含两个 b 的字符串的集合。
37. $\{a, b\}$ 上偶数个 a , 或者奇数个 b 构成的字符串的集合。
38. $\{a, b\}$ 上偶数个 a 和偶数个 b 构成的字符串的集合。这里有点小技巧: 构造这个表达式的策略参见第 6 章。
39. 使用表 2-1 中给出的正则表达式恒等式来构造下面的恒等式:
- $(ba)^+(a^*b^* \cup a^*) = (ba)^+ba^*(b^* \cup \lambda)$
 - $b^+(a^*b^* \cup \lambda)b = b(b^*a^* \cup \lambda)b^+$
 - $(a \cup b)^+ = (a \cup b)^*b^+$
 - $(a \cup b)^+ = (a^* \cup ba^*)^+$
 - $(a \cup b)^+ = (b^*(a \cup \lambda)b^*)^+$
40. 在 2.4 节的凯撒文件中, 搜索分别满足下列扩展的正则表达式的输出。
- $[Cc]$
 - $[K-Z]$
 - $\backslash < [a-z] \{6\} \backslash >$
 - $\backslash < [a-z] \{6\} \backslash > \backslash < [a-z] \{7\} \backslash >$
41. 设计一个扩展的正则表达式来完成对地址的搜索。针对这个练习, 一个地址要包括
- 数字,
 - 街道名字, 和
 - 街道类型标识符或是缩写。
- 你的模式应该匹配形如 1428 Elm Street, 51095 Tobacco Rd. 和 1600 Pennsylvania Avenue 这样形式的地址。如果你的正则表达式不能够识别所有可能的地址, 你也不需要担心。

参考文献注释

- [61] 正则表达式是由 Kleene [1956] 出于研究神经网络的目的提出的。McNaughton 和 Yamada
[62] [1960] 证明正则集合在并和补操作下是封闭的。正则表达式的代数学公理化可参考 Salomaa [1966]。

第二部分

文法、自动机和语言

语言的文法指定了语言中所允许的字符串的形式。在第2章中，集合论操作和递归定义都用来生成语言的字符串。这些构造字符串的工具尽管简单，但却足够用来确定构成字符串的元素的数量和顺序的简单限制。我们在这里要介绍一种基于规则的方法来定义和生成语言的字符串。这种定义语言的方法在语言学和计算机科学中都有它的本源：语言学试图形式化地描述自然语言，而计算机科学需要提供准确、无二义性的高级语言定义。使用语言学的术语，我们称字符串生成系统为文法。

在第3章中，我们将介绍两种文法：正则文法和上下文无关文法。一族文法是通过规则和它们适用的条件的形式来定义。规则给出了字符串转换的方法，语言的字符串是通过应用这一系列的规则来生成的。由规则提供的灵活性已经证明了可以很好地适用于定义编程语言的语法。描绘编程语言 Java 的文法给出了构造几种常见的编程语言的上下文无关定义。

利用字符串的产生过程来定义语言，我们就可以把关注点转移到机械验证上了：一个字符串是否满足期望的条件，或者是否匹配了期望的模式。确定型有限自动机家族是一系列功能渐强的抽象机中的第一个。我们将把这些抽象机用于模式匹配和语言定义中。之所以把机器看成抽象的，是因为我们不需要考虑它们的硬件和软件实现。相反，我们更关心这些机器的计算能力。抽象机的输入是字符串，计算的结果表明了其对输入串的接收能力。机器的语言就是可以被机器的计算过程接受的字符串的集合。

确定型有限自动机是只读型机器，它会根据当时机器的状态和待处理的输入字符来确定要执行的指令。有限自动机有很多应用，其中包括计算机程序的词法分析、数字电路设计、文本搜索和模式识别。Kleene 理论证明了有限自动机接收的语言只是那些可以用正则表达式描述并能由正则文法生成的语言。一个更强大的只读机器就是下推自动机。它在有限自动机的基础上增加了一个栈内存。这个新添加的内存允许下推自动机接收上下文无关文法。

文法所生成的语言和文法被机器接受之间的对应关系是这本书的核心主题。机器和文法之间的关系将会在第3部分的无限制文法家族和图灵机中继续介绍。正则文法、上下文无关文法和无限制文法都是第10章中的乔姆斯基文法层次中的成员。

第3章 上下文无关文法

本章我们将给出产生语言字符串的基于规则的方法。借用自然语言中的术语，我们称一个语法正确的字符串为语言的**句子**（sentence）。英语的小子集可以用来解释字符串生成过程的构成成分。我们缩小的语言的字母表示集合 $a, the, John, Jill, hamburger, car, drives, eats, slowly, frequently, big, juicy, brown$ 。字母表中的元素称为语言的**终结符**（terminal symbols）。大写、标点和书面语言的其他重要特点在这个例子中都可以忽略。

句子生成程序可以用来构造字符串 *John eats a hamburger* 和 *Jill drives frequently*。形如 *Jill* 和 *car* *John slowly* 的字符串形式是无法由这个过程产生的。在构造句子的过程中还可以使用其他符号，来强调语言的语法限制。这些中间符号，叫做变量（variables）或非终结符（nonterminals），使用尖括号〈〉来突出表示。

因为生成程序构造了句子,所以初始变量也叫做 *sentence*。句子的生成包括把变量替换成某种特别的字符串形式,使用一个转换规则集合可以完成语法的正确替换。对于变量 *(sentence)* 两个的可能规则是

1. $\langle \text{sentence} \rangle \rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle$
2. $\langle \text{sentence} \rangle \rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb} \rangle \langle \text{direct-object-phrase} \rangle$

规则1的非形式化解释是句子可以用一个名词短语,后面跟着一个动词短语来构成。当然,在这点上,变量〈*noun-phrase*〉和〈*verb-phrase*〉都没有定义。第二个规则给出了句子的替换定义,即:一个名词短语,后面跟着一个带有指定对象短语的动词。多种转换的存在暗示了语法正确的句子可以有不同的形式。

名词短语可以包括一个特有名词或者常用名词。常用名词指的是带有限定词的,而特有名词指的是没有限定词的。英语语法的特点定义为规则3和规则4。

产生名词和动词短语的变量规则定义如下。我们依次列出了规则的右侧，而不是重复列出规则的左部。给这些规则编号并不是生成过程的特点，只不过是方便而已。

3. $\langle \text{noun-phrase} \rangle \rightarrow \langle \text{proper-noun} \rangle$
4. $\rightarrow \langle \text{determiner} \rangle \langle \text{common-noun} \rangle$
5. $\langle \text{proper-noun} \rangle \rightarrow \text{John}$
6. $\rightarrow \text{Jill}$
7. $\langle \text{common-noun} \rangle \rightarrow \text{car}$
8. $\rightarrow \text{hamburger}$
9. $\langle \text{determiner} \rangle \rightarrow a$
10. $\rightarrow the$
11. $\langle \text{verb-phrase} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{adverb} \rangle$
12. $\rightarrow \langle \text{verb} \rangle$
13. $\langle \text{verb} \rangle \rightarrow \text{drives}$
14. $\rightarrow \text{eats}$
15. $\langle \text{adverb} \rangle \rightarrow \text{slowly}$
16. $\rightarrow \text{frequently}$

除了 $\langle \text{direct-object-phrase} \rangle$ 以外，每个变量的规则都给出了。

应用规则的使用, 可以把一个字符串转化成另一个字符串。这种转化指的是把 \rightarrow 左侧的变量替换成它右侧的部分。在句子的产生过程中, 反复地应用规则从而把变量 $\langle sentence \rangle$ 转化成其他终结符

号字符串。

例如, 句子 *Jill drives frequently* 是使用下面的转化完成的。

符号 \Rightarrow 用来指定一个规则应用, 读作“推导”。

右侧的列给出了变换中使用的规则的序号。当所有的变量都从待推导的字符串中去掉时, 推导就结束了。结果的字符串, 仅仅包括终结符号, 就是语言的句子。由变量 $\langle sentence \rangle$ 开始推导得出的终结字符串的集合就是使用我们例子中的规则产生的语言。

推 导	应用规则
$\langle sentence \rangle \Rightarrow \langle noun\text{-}phrase \rangle \langle verb\text{-}phrase \rangle$	1
$\Rightarrow \langle proper\text{-}noun \rangle \langle verb\text{-}phrase \rangle$	3
$\Rightarrow \langle Jill \rangle \langle verb\text{-}phrase \rangle$	6
$\Rightarrow \langle Jill \rangle \langle verb \rangle \langle adverb \rangle$	11
$\Rightarrow \langle Jill \rangle \langle drives \rangle \langle adverb \rangle$	13
$\Rightarrow \langle Jill \rangle \langle drives \rangle \langle frequently \rangle$	16

66

为了使我们的规则更加完整, $\langle direct\text{-}object\text{-}phrase \rangle$

的转化必须给出。在给出这个规则之前, 我们必须决

定要产生的字符串的形式。在我们的语言中, 我们允许在直接对象之前存在任何数量的形容词, 包括重复的。这就要求存在一个能够产生下列字符串的规则集合

John eats a hamburger
John eats a big hamburger
John eats a big juicy hamburger
John eats a big brown juicy hamburger
John eats a big big brown juicy hamburger

因为形容词可能重复出现, 所以文法的规则必须能够产生任意长度的字符串。递归定义的使用使得由有限规约生成无穷集合的元素成为可能。在这个例子之后, 递归将被引入到字符串生成过程当中, 即, 引入规则

17. $\langle adjective\text{-}list \rangle \rightarrow \langle adjective \rangle \langle adjective\text{-}list \rangle$

18. $\rightarrow \lambda$

19. $\langle adjective \rangle \rightarrow big$

20. $\rightarrow juicy$

21. $\rightarrow brown$

$\langle adjective\text{-}list \rangle$ 定义遵循了标准递归模式。规则 17 递归地定义了 $\langle adjective\text{-}list \rangle$, 而规则 18 提供了递归定义的基础。规则 18 右部的 λ 指的是应用这条规则, 把 $\langle adjective\text{-}list \rangle$ 替换成空串。重复使用规则 17 就产生了一系列的形容词。 $\langle adjective\text{-}object\text{-}phrase \rangle$ 的规则是使用 $\langle adjective\text{-}list \rangle$ 来构造的。

22. $\langle direct\text{-}object\text{-}phrase \rangle \rightarrow \langle adjective\text{-}list \rangle \langle proper\text{-}noun \rangle$

23. $\rightarrow \langle determiner \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$

句子 *John eats a big juicy hamburger* 可以通过使用下面的一系列规则推导而成

推 导	应用规则
$\langle sentence \rangle \Rightarrow \langle noun\text{-}phrase \rangle \langle verb \rangle \langle direct\text{-}object\text{-}phrase \rangle$	2
$\Rightarrow \langle proper\text{-}noun \rangle \langle verb \rangle \langle direct\text{-}object\text{-}phrase \rangle$	3
$\Rightarrow \langle John \rangle \langle verb \rangle \langle direct\text{-}object\text{-}phrase \rangle$	5
$\Rightarrow \langle John \rangle \langle eats \rangle \langle direct\text{-}object\text{-}phrase \rangle$	14
$\Rightarrow \langle John \rangle \langle eats \rangle \langle determiner \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	23
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	9
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle adjective \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	17
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle big \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	19
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle big \rangle \langle adjective \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	17
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle big \rangle \langle juicy \rangle \langle adjective\text{-}list \rangle \langle common\text{-}noun \rangle$	20
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle big \rangle \langle juicy \rangle \langle common\text{-}noun \rangle$	18
$\Rightarrow \langle John \rangle \langle eats \rangle \langle a \rangle \langle big \rangle \langle juicy \rangle \langle hamburger \rangle$	8

67

句子的产生过程是这些规则严格的函数。字符串 *the car eats slowly* 是这种语言的句子，因为它具有规则 1 中 $\langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle$ 的形式。这就解释了语法和语义的重要区别，句子的产生过程只考虑推导的字符串中的形式，而不考虑它与终结符号相关的潜在意义。

根据规则 3 和规则 4，名词短语包括特定名词或前面带限定词的通用名词。变量 $\langle \text{adjective-list} \rangle$ 可以添加到 $\langle \text{noun-phrase} \rangle$ 规则当中，允许形容词来修改名词。

3'. $\langle \text{noun-phrase} \rangle \rightarrow \langle \text{adjective-list} \rangle \langle \text{proper-noun} \rangle$

4'. $\rightarrow \langle \text{determiner} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$

使用这种修改的规则，字符串 *big John eats frequently* 就可以由变量 $\langle \text{sentence} \rangle$ 推导而得

3.1 上下文无关文法和语言

我们现在定义一个形式化系统——上下文无关文法，它可以用来生成一种语言的字符串。自然语言的例子用来描述使用上下文无关文法生成的字符串的构成成分和具有的特点

定义 3.1.1 上下文无关文法 (context-free grammar) 是一个四元组 (V, Σ, P, S) ，其中， V 是变量的有限集合， Σ (字母表) 是终结符的有限集合， P 是规则的有限集合， S 区别于 V 中的元素，称为开始符，并且还假设集合 V 和 Σ 是不相交的。

规则 (rule) 写成 $A \rightarrow w$ 的形式，其中， $A \in V$ 并且 $w \in (V \cup \Sigma)^*$ 。这种形式的规则称作 A 规则，指的是变量在左边的。因为空串属于 $(V \cup \Sigma)^*$ ，所以 λ 可能出现在规则的右侧。形如 $A \rightarrow \lambda$ 的规则称为空规则 (null)，或 λ -规则 (λ -rule)。

斜体用来表示上下文无关文法的变量和终结符。终结符用字母表中前面的字母的小写形式表示，比如 a, b, c, \dots 。遵循字符串定义的规则，没有或有上角标的字母 p, q, u, v, w, x, y, z 表示 $(V \cup \Sigma)^*$ 中的任意成员。变量使用大写字母表示。像自然语言的例子，变量是文法中的非终结符号 (nonterminal symbols)。

文法用来产生规定字母表上具有正确形式的字符串。生成过程的基础步骤包括使用规则来转换字符串。把 $A \rightarrow w$ 应用到 uAv 中的变量 A 中，就会产生字符串 uwv 。这就可以表示成 $uAv \Rightarrow uwv$ 。前缀 u 和后缀 v 定义了变量 A 出现的上下文 (context)。本章介绍的文法叫做上下文无关，因为规则是通用的。 A 规则可以用到变量 A 上，当且仅当变量 A 出现的时候，并且只可以用在它出现的地方。上下文对于这个规则的应用没有任何限制。

如果应用有限条规则可以把 v 转化成 w ，那么字符串 w 可以由 v 推导出。即，如果使用文法的规则可以构造下面

$$v \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n = w$$

这一系列的转化。从 v 推导出 w 表示成 $v \Rightarrow^* w$ 。应用有限步规则构造的从 v 推导而得的字符串的集合可以递归地定义。

定义 3.1.2 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法，并且 $v \in (V \cup \Sigma)^*$ 。由 v 推导 (derivable) 的字符串的集合可以递归地定义为：

i) 基础步骤： v 可以由 v 推导出。

ii) 递归步骤：如果 $u = xAy$ 可以由 v 推导出，并且 $A \rightarrow w \in P$ ，那么 xwy 就可以由 v 推导出。

iii) 封闭：字符串可以由 v 推导出，仅当它可以由 v 经过有限步归纳构造而成。

值得注意的是，规则的定义中使用的是 \rightarrow 符号，而不是 \Rightarrow 符号。 \Rightarrow 表示可推导出，而 \Rightarrow^* 表示使用至少一步可以推导出。推导的长度就是应用规则的数目。从 v 到 w 的长度为 n 的推导表示成 $v \Rightarrow^n w$ 。当考查多于一种文法的时候，我们使用 $v \xRightarrow{G} w$ 表示应用了文法 G 的推导规则。

语言是定义在字母表上的字符串的集合。文法包含字母表和产生字符串的方法。这些字符串既包括变量，也包括终结符。设想 $\langle \text{sentence} \rangle$ 在自然语言例子中的角色，文法的开始符初始化了产生可接受字符串的过程。文法 G 的语言就是由这个开始符推导得到的终结符的集合。我们下面给出

相应的定义。

定义 3.1.3 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法。

- i) 如果 G 中存在推导 $S \Rightarrow^* w$, 那么字符串 $w \in (V \cup \Sigma)^*$ 就是 G 的句型 (sentential form)。
- ii) 如果 G 中存在推导 $S \Rightarrow^* w$, 那么字符串 $w \in \Sigma^*$ 是 G 的句子 (sentence)。
- iii) G 的语言 (language), 记成 $L(G)$, 是指集合 $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$ 。

句型是由文法的开始符推导得到的字符串。回顾自然语言的例子, 推导

$$\begin{aligned} \langle \text{sentence} \rangle &\Rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle \\ &\Rightarrow \langle \text{proper-noun} \rangle \langle \text{verb-phrase} \rangle \\ &\Rightarrow \text{Jill} \langle \text{verb-phrase} \rangle \end{aligned}$$

表明 $\text{Jill} \langle \text{verb-phrase} \rangle$ 是文法的句型。它不是句子, 因为包含变量, 但是它具有句子的形式。句子是只包含终结符的句型。文法的语言包含由文法产生的句子。如果字母表 Σ 上的字符串可以由上下文无关文法产生, 那么这些字符串构成的集合就是上下文无关语言 (context-free language)。

递归的使用对于构造任意长度的字符串, 以及由无穷多的字符串组成的语言是必要的。递归是通过文法引入到规则当中的。形如 $A \Rightarrow^* uAv$ 的规则是递归的 (recursive), 因为它使用 A 自身来定义 A 。形如 $A \Rightarrow^* Av$ 和 $A \Rightarrow^* uA$ 的规则分别称为是左递归的 (left-recursive) 和右递归的 (right-recursive)。它们的名字暗示了它们在规则中递归的位置。

因为递归规则的重要性, 所以我们检查了反复使用的递归规则 $A \Rightarrow^* aAb$, $A \Rightarrow^* aA$, $A \Rightarrow^* Ab$ 和 $A \Rightarrow^* AA$ 生成的字符串形式。

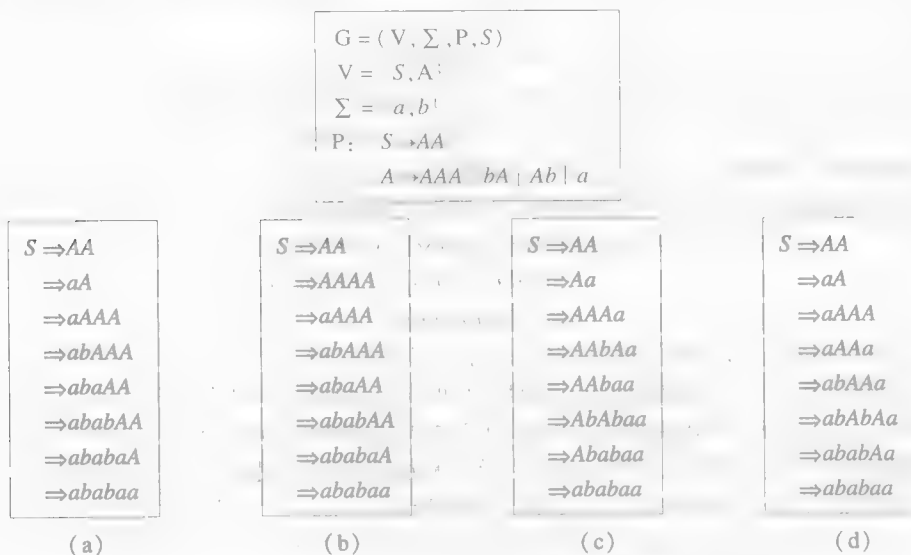
$A \Rightarrow^* aAb$	$A \Rightarrow^* aA$	$A \Rightarrow^* Ab$	$A \Rightarrow^* AA$
$\Rightarrow^* aAb$	$\Rightarrow^* aA$	$\Rightarrow^* Ab$	$\Rightarrow^* AAA$
$\Rightarrow^* aaAbb$	$\Rightarrow^* aaA$	$\Rightarrow^* Abb$	$\Rightarrow^* AAAAA$
$\Rightarrow^* aaaAbbb$	$\Rightarrow^* aaaA$	$\Rightarrow^* Abbb$	$\Rightarrow^* AAAAAA$
\vdots	\vdots	\vdots	\vdots

推导中使用规则 $A \Rightarrow^* aAb$, 可以生成一定数量的 a , 然后后面紧跟着相同数目的 b 。这种形式的规则对于生成包含成对符号的字符串是必要的, 比如左右括号。右递归规则 $A \Rightarrow^* aA$ 在变量 A 之前产生任意数目的 a , 而左递归规则 $A \Rightarrow^* Ab$ 在变量 A 后面生成任意数目的 b 。每次应用规则 $A \Rightarrow^* AA$ —— 这个规则既是左递归的, 又是右递归的 —— 就会增加一个 A 。递归规则的重复使用可以利用另外一个 A 规则来终止。

如果存在推导 $A \Rightarrow^* uAv$, 那么变量 A 是递归的 (recursive)。形如 $A \Rightarrow^* w \Rightarrow^* uAv$ 的推导, 其中, A 不属于 w , 是间接递归的 (indirectly recursive)。值得注意的是根据间接递归, 变量 A 可以是递归的, 即使不存在递归 A 规则。

图 3-1 给出了文法 G , 这种文法产生了包含正偶数个 a 的字符串的语言。这个文法的规则是使用速记 $A \Rightarrow^* uv$ 来简化 $A \Rightarrow^* u$ 和 $A \Rightarrow^* v$ 。竖直线“ $|$ ”读作“或”。图 3-1 给出了终结符 $ababaa$ 的四种不同推导。推导的定义允许字符串中的任何变量的转化。推导 (a) 和 (b) 中应用的每个规则转化的对象的都是字符串中从左到右出现的第一个变量。具有这种性质的推导叫做最左的 (leftmost) 推导 (c) 是最右的 (rightmost), 因为每次规则都应用到最右边的变量上。这些推导表明上下文无关文法的字符串存在多种推导。

图 3-1 展示了上下文无关文法的推导的灵活性。推导的本质特点不是应用规则的顺序, 而是把每个变量转化成终结字符串方式。转化的过程使用推导和分析树的形式图形化地描述出来。树的结构体现了应用到每个变量上的规则, 而不是给出了应用规则的顺序。推导树的叶子展示了树表示的推导的结果。

图 3-1 G 中的 $ababaa$ 的推导例子

定义 3.1.4 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法, 并且 $S \Rightarrow^* w$ 是 G 中的推导 $S \Rightarrow^* w$ 的推导树 (derivation tree) DT 可以按照下面循环构造:

71

i) 用根节点 S 来初始化 DT。

ii) 如果 $A \rightarrow x_1 x_2 \cdots x_n$ 是应用于推导字符串 uAv 中的规则, 并且 $x_i \in (V \cup \Sigma)$, 那么就在树中增加 A 的子节点 x_1, x_2, \cdots, x_n 。

iii) 如果 $A \rightarrow \lambda$ 是应用于推导字符串 uAv 的规则, 那么就在树中 A 的子节点增加 λ 。

叶子的顺序也遵循这个循环过程。起初, 惟一的叶子是 S , 顺序一目了然。当应用规则 $A \rightarrow x_1 x_2 \cdots x_n$ 来生成 A 的子节点时, 每个 x_i 都成了叶节点, 并且用序列 x_1, x_2, \cdots, x_n 来替换 A 。规则 $A \rightarrow \lambda$ 用来把 A 替换成空串。图 3-2 跟踪了图 3-1 推导 (a) 中对应的树的构造过程。叶子的顺序是和每棵树一起给出的。

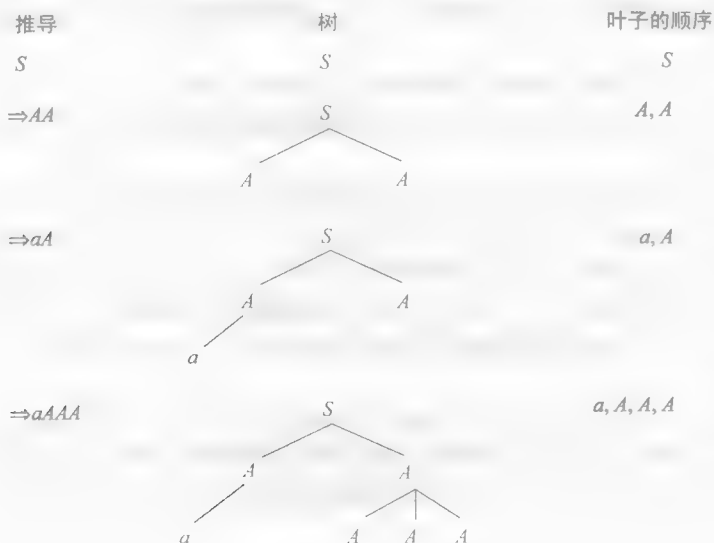


图 3-2 推导树的构造

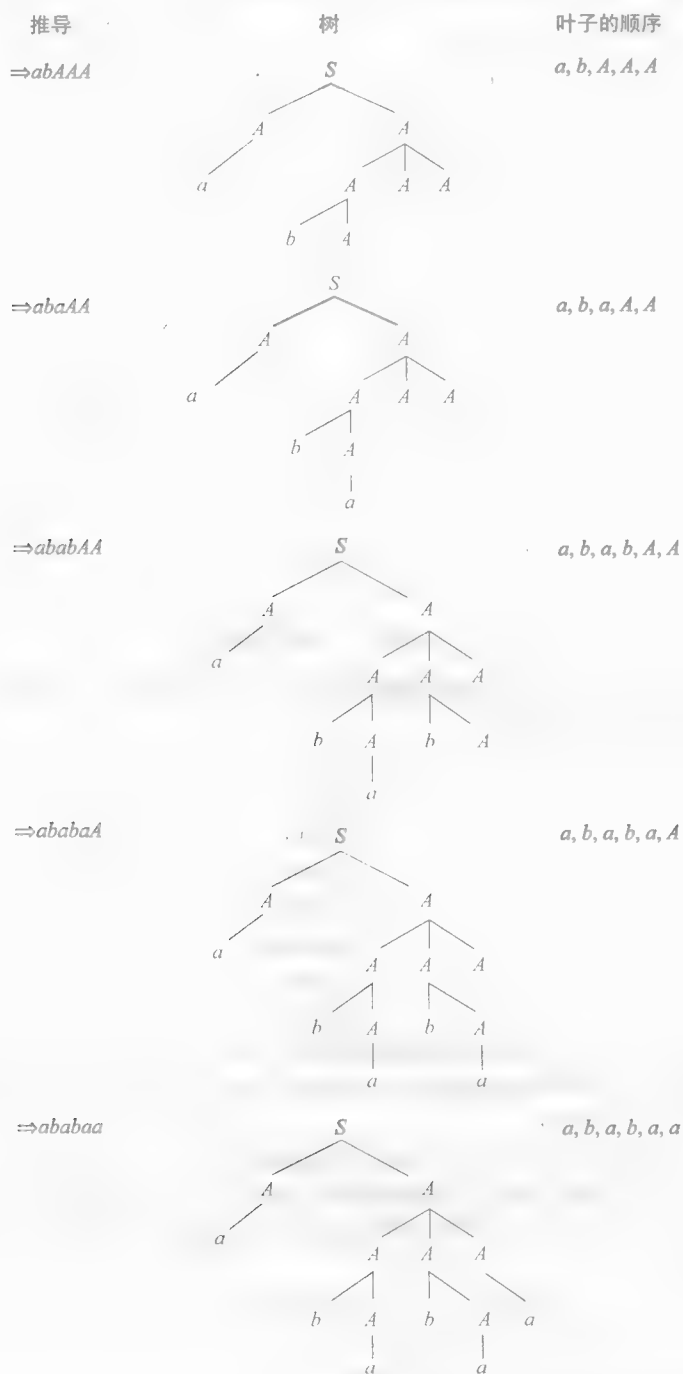


图3-2 (续)

推导树上的叶子的顺序独立于产生树的推导。上述循环过程提供的顺序等同于1.8节中关系LEFTOF中给出的叶子的顺序。推导树的边界就是推导产生的字符串。

图3-3给出了图3-1中的每个推导对应的推导树。推导(a)和(d)产生的树是同一个,这意味着每个变量都转化成同一模式的终结字符串。这些推导的唯一区别就是规则使用的顺序。

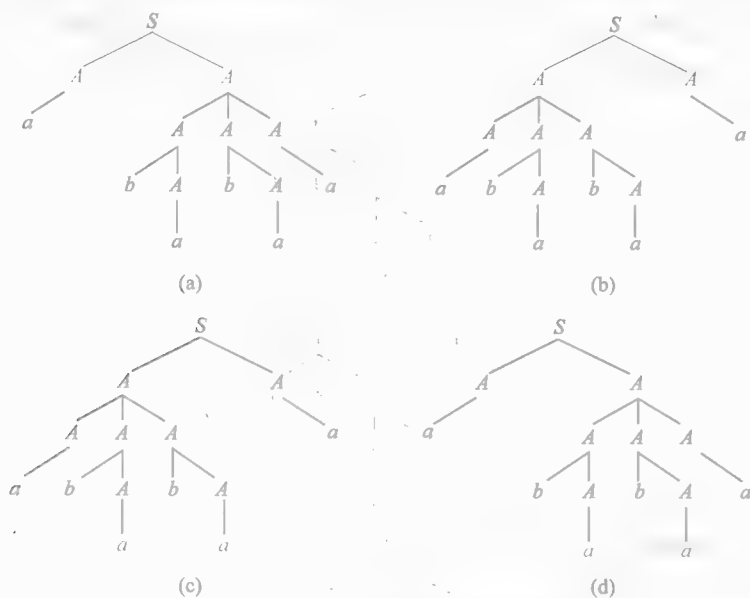


图 3-3 与图 3-1 中推导对应的树

推导树可以用来产生若干个推导, 这些推导都是产生同一个字符串的。应用到变量 A 的规则可以用来重构树中 A 的子节点。最右推导:

$$\begin{aligned}
 S &\Rightarrow AA \\
 &\Rightarrow AAAA \\
 &\Rightarrow AAAa \\
 &\Rightarrow AAbAa \\
 &\Rightarrow AAbaa \\
 &\Rightarrow AbAbaa \\
 &\Rightarrow Ababaa \\
 &\Rightarrow ababaa
 \end{aligned}$$

是从图 3-3 中的推导树中获得的。值得注意的是这些推导不同于图 3-1 中的最右推导 (c)。在后者的推导中, 字符串 AA 的第二个变量使用规则 $A \rightarrow a$ 进行转化, 而 $A \rightarrow AAA$ 用于接下来的推导。这两棵树图形化地解释了这两种不同的转化。

正如我们看到的, 规则的上下文无关应用使得构造推导的过程中存在大量灵活的技巧。引理 3.1.5 证明了推导可以被字符串中的每个变量拆分成多个子推导。推导可以递归地定义, 推导的长度是有限的, 但是却是没有界限的。因此, 我们可以使用数学推理来证明一种性质, 即对于一个给定字符串的所有推导都成立。

引理 3.1.5 已知 G 是上下文无关文法, 并且 $v \xRightarrow{n} w$ 是 G 中的推导, 其中, v 可以写成

$$v = w_1 A_1 w_2 A_2 \cdots w_k A_k w_{k+1},$$

并且 $w_i \in \Sigma^*$ 。于是存在字符串 $p_i \in (\Sigma \cup V)^*$ 满足

- i) $A_i \xRightarrow{t_i} p_i$
- ii) $w = w_1 p_1 w_2 p_2 \cdots w_k p_k w_{k+1}$
- iii) $\sum_{i=1}^k t_i = n$

证明: 对从 v 到 w 的推导长度进行归纳证明。

基础步骤: 基础包括形如 $v \Rightarrow w$ 的推导。在这种情况下, $w = v$ 并且每个 A_i 都等于对应的 p_i 。期望

的推导具有 $A_i \Rightarrow^0 p_i$ 的形式。

归纳假设: 假设所有的推导 $v \Rightarrow^n w$ 都可以拆分成: 由 A_i 开始, 变量是 v , 从 v 到 w 的长度为 n 的推导。

归纳步骤: 已知 $v \Rightarrow^{n+1} w$ 是 G 中的推导, 并且

$$v = w_1 A_1 w_2 A_2 \cdots w_k A_k w_{k+1},$$

其中 $w_i \in \Sigma^*$ 。这个推导可以写成 $v \Rightarrow u \Rightarrow w$ 。这就把原始的推导化简成一条简单规则的应用和一个长度为 n 的推导。后者根据归纳假设是成立的。

推导中应用的第一个规则是 $v \Rightarrow w$, 把 v 中的一个变量变形, 记作 A_j , 形如规则

$$A_j \rightarrow u_1 B_1 u_2 B_2 \cdots u_m B_m u_{m+1},$$

其中, 每个 $u_i \in \Sigma^*$ 。字符串 u 是通过使用 A_j 规则把 v 右侧的 A_j 替换而成的。进行这样的替换, u 可以写成

$$w_1 A_1 \cdots A_{j-1} w_j u_1 B_1 u_2 B_2 \cdots u_m B_m u_{m+1} w_{j+1} A_{j+1} \cdots w_k A_k w_{k+1}.$$

因为 w 是使用 n 步规则由 u 推导而成, 因此归纳假设保证存在字符串 $p_1, \dots, p_{j-1}, q_1, \dots, q_m$ 和 p_{j+1}, \dots, p_k 满足

i) 对于 $i=1, \dots, j-1, j+1, \dots, k$ 有 $A_i \Rightarrow^0 p_i$

对于 $i=1, \dots, m$ 有 $B_i \Rightarrow^0 q_i$

ii) $w = w_1 p_1 w_2 \cdots p_{j-1} w_j u_1 q_1 u_2 \cdots u_m q_m u_{m+1} w_{j+1} p_{j+1} \cdots w_k p_k w_{k+1}$ 并且

iii) $\sum_{i=1}^{j-1} t_i + \sum_{i=j+1}^k t_i + \sum_{i=1}^m s_i = n$

把规则 $A_j \rightarrow u_1 B_1 u_2 B_2 \cdots u_m B_m u_{m+1}$ 和推导 $B_i \Rightarrow^0 q_i$ 相结合, 我们会获得推导

$$A_j \Rightarrow u_1 q_1 u_2 q_2 \cdots u_m q_m u_{m+1} = p_j.$$

这个推导的长度是由 B 开始的推导的总长度加上推导 $A_i \Rightarrow^0 p_i (i=1, \dots, k)$ 提供了由 v 到 w 的推导的理想分解。

引理 3.1.5 表明了上下文无关文法推导的灵活性和模块性。每个复杂的推导都可以分解成成员变量的子推导。复杂语言设计的模块性体现在使用变量定义较小的以及更容易管理的语言的子集上。这些独立定义的子语言和其他的规则相结合, 就产生了整个语言的语法。

3.2 文法和语言的例子

上下文无关文法可以用来产生语言。形式语言, 和计算机语言、自然语言类似, 都有字符串需要满足的要求, 从而保证它们的语法正确。这些语言的文法必须能够准确地产生理想的字符串, 而不是其他的字符串。我们可以使用两种普通的方法来帮助理解文法和语言之间的关系。一种是首先构造语言的非形式化规约, 然后构造产生这种语言的文法。这是遵循编程语言设计的方法——选择好语法, 然后语言设计者产生一个定义正确形式字符串的规则集合。与之相反, 另一种方法是先构造文法的规则, 然后分析它们来确定语言的字符串的形式。这就是在检查计算机程序代码的语法时经常使用的方法。编程的语法是使用一个文法规则集合来指定的, 比如附录 IV 中给出的编程语言 Java 的定义。如果代码可以由文法中正确的变量推导得出, 那么常数、标识符、语句和整个程序的语法就都是正确的。

起初, 确定字符串和规则之间的关系看起来很困难。但随着经验的增长, 你就可以识别出字符串中经常出现的模式, 以及产生它们的规则。本节的目的就是通过分析例子来帮助你更直观地理解由上下文无关文法定义的语言。

在每个例子中, 我们都是通过列举文法的规则来定义文法。文法的变量和终结符就是规则中出现的那些符号。变量 S 是每种文法的开始符。

例 3.2.1 已知 G 是按照下面的规则给出的文法

$$S \rightarrow aSa \mid aBa$$

$$B \rightarrow bB \mid b.$$

那么 $L(G) = \{a^n b^m a^n \mid n > 0, m > 0\}$ 。规则 $S \rightarrow aSa$ 递归地构造了首尾具有相同数目的 a 的字符串。递归过程的终止是通过使用规则 $S \rightarrow aBa$ ，从而保证至少有一个处于首位的 a 和处于尾部的 a 。递归 B 规则产生了任意数目的 b 。为了从字符串中去掉变量 B ，并且获得该语言的句子，必须应用规则 $B \rightarrow b$ ，从而使得产生的字符串中至少存在一个 b 。□

例 3.2.2 语言 $\{a^n b^m c^m d^n \mid n \geq 0, m > 0\}$ 中开头的 a 和结尾处的 d 的数目之间的关系暗示了需要一个递归规则才能产生它们。 b 和 c 的个数也一样。文法的推导

$$S \rightarrow aSdd \mid A$$

$$A \rightarrow bAc \mid bc$$

产生了从外到内模式的字符串。 S 规则产生了同样个数的 a 和 d ，而 A 规则产生了同样个数的 b 和 c 。应用规则 $A \rightarrow bc$ 终止递归过程，从而保证字符串 bc 出现在语言的每个字符串中。□

例 3.2.3 回想如果 $w = w^R$ ，则字符串 w 是回文。构造产生 a, b 上的回文的集合的文法。文法的规则模仿了练习 2.1.2 中给出的回文的递归定义。回文集合的基础包括字符串 λ 、 a 和 b 。 S 规则

$$S \rightarrow a \mid b \mid \lambda$$

直接产生了这些字符串。定义的递归部分向已有的回文的两侧增加相同的字符。规则

$$S \rightarrow aSa \mid bSb$$

完成了递归生成过程。□

例 3.2.4

$$S \rightarrow aSb \mid aSbb \mid \lambda$$

的第一个递归规则为每个 a 产生一个尾部的 b ，第二个规则为每个 a 产生两个 b 。因此，对于每个 a ，至少有一个，至多有两个 b 。这个文法的语言是 $\{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$ 。□

例 3.2.5 考查文法

$$S \rightarrow abScB \mid \lambda$$

$$B \rightarrow bB \mid b.$$

递归 S 规则产生了同样数目的 ab 和 cB 。 B 规则产生了 b^+ 。在推导中， B 的每次出现都会相伴产生不同数目的 b 。例如，在推导

$$\begin{aligned} S &\Rightarrow abScB \\ &\Rightarrow ababScBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcBcBb, \end{aligned}$$

中， B 的第一次出现产生了一个 b ，第二次出现产生了 bb 。这个文法的语言是集合 $\{(ab)^n (cb^{m_i})^n \mid n \geq 0, m_i > 0\}$ 。 m_i 的上角标表示每个 B 产生的 b 的数目，因为当 $i \neq j$ 时， b^{m_i} 不必等于 b^{m_j} 。□

例 3.2.6 已知 G_1 和 G_2 是文法

$$G_1: S \rightarrow AB \quad G_2: S \rightarrow aS \mid aA$$

$$A \rightarrow aA \mid a \quad A \rightarrow bA \mid \lambda.$$

$$B \rightarrow bB \mid \lambda$$

这两种文法都可以产生语言 $a^* b^*$ 。 G_1 中的 A 规则提供了产生 a 的非空串的标准方法。使用 λ 规则可以终止推导过程，从而允许没有 b 的出现。文法 G_2 中的规则从左到右地构造了字符串 $a^* b^*$ 。□

例 3.2.7 文法 G_1 和 G_2 生成 $\{a, b\}$ 上只包含两个 b 的字符串。即：文法的语言是 $a^* ba^* ba^*$ 。

$$\begin{aligned} G_1: S \rightarrow AbAbA \quad G_2: S \rightarrow aS \mid bA \\ A \rightarrow aA \mid \lambda \quad A \rightarrow aA \mid bC \\ C \rightarrow aC \mid \lambda. \end{aligned}$$

G_1 要求只有两个变量, 因为 a^* 的三个实例是使用同一个 A 规则产生的。第二个文法从左到右地构造了字符串, 并使用每个不同的变量产生一系列的 a 。□

例 3.2.8 进一步修改例 3.2.7 中的文法, 使得字符串中至少有两个 b 。

$$\begin{aligned} G_1: S \rightarrow AbAbA \quad G_2: S \rightarrow aS \mid bA \\ A \rightarrow aA \mid bA \mid \lambda \quad A \rightarrow aA \mid bC \\ C \rightarrow aC \mid bC \mid \lambda. \end{aligned}$$

在 G_1 中, 根据 S 规则, 任何字符串都可以生成在两个 b 之前、之间和之后。 G_2 中的推导使用规则 $S \rightarrow bA$ 产生了第一个 b , 使用 $A \rightarrow bC$ 产生了第二个 b 。使用 C 规则结束推导, 从而可以生成 a 和 b 的任何字符串。□

这两种文法产生的语言称作是等价的 (equivalent)。例 3.2.6、例 3.2.7 和例 3.2.8 显示等价文法可以产生推导过程迥异的语言的字符串。在后面的章节中, 我们将看到这些规则存在特殊的形式, 这些规则使得检查字符串的语法是否正确变得更加容易了。

例 3.2.9 给出产生 $\{a, b\}^*$ 上包含偶数个 b 的字符串的语言的文法。相应的策略可以用来构造长度可以被 3、被 4 等整除的字符串。变量 S 和 O 是计数器。每当产生偶数个终结符时, S 就出现在句型当中。 O 记录了奇数个终结符的出现。

$$\begin{aligned} S &\rightarrow aO \mid bO \mid \lambda \\ O &\rightarrow aS \mid bS \end{aligned}$$

应用 $S \rightarrow \lambda$ 完成了终结字符串的推导, 反复应用 S 和 O 规则直到应用这条。□

例 3.2.10 已知 L 是 $\{a, b\}^*$ 上包含偶数个 b 的所有字符串。生成 L 的文法

$$\begin{aligned} S &\rightarrow aS \mid bB \mid \lambda \\ B &\rightarrow aB \mid bS \mid bC \\ C &\rightarrow aC \mid \lambda \end{aligned}$$

结合了前面例子中的技巧, 包括例 3.2.9 中生成偶数个 b 的技巧和例 3.2.7 给出的生成任意个 a 的技巧。去掉所有包含 C 的规则, 就得到了生成 L 的另一个文法。□

例 3.2.11 练习 2.38 要求构造 $\{a, b\}^*$ 上由包含偶数个 a 和偶数个 b 的字符串构成的语言的正则表达式。值得注意的是, 在那个时候构造这种语言的正则表达式还是比较复杂的。按照规则生成的字符串所提供的灵活性, 使得构造这种语言的上下文无关文法更加直接。一些变量被选出来表示推导出的字符串中的部分 a 和 b 的个数。文法中变量的解释如下:

变 量	解 释
S	偶数 a , 偶数 b
A	偶数 a , 奇数 b
B	奇数 a , 偶数 b
C	奇数 a , 奇数 b

应用规则把一个终结符添加到一个推导出的字符串中, 并更新变量使它反映最新的状态。文法的规则是

$$\begin{aligned} S &\rightarrow aB \mid bA \mid \lambda \\ A &\rightarrow aC \mid bS \\ B &\rightarrow aS \mid bC \\ C &\rightarrow aA \mid bB. \end{aligned}$$

当出现变量 S 时, 推导出的字符串就具有偶数个 a 和偶数个 b 。应用 $S \rightarrow \lambda$ 就可以把句型中的变量去掉, 从而生成满足语言规约的字符串。□

例 3.2.12 文法的规则是用来描述语言的字符串结构的。这种结构确保字母表中元素的某种组合的存在或不存在。我们构造字母表 $\{a, b, c\}$ 上的文法, 使得构造得到的语言的字符串不包含子串 abc 。变量用来确定现在的推导距离生成字符串 abc 还有多远。

$$S \rightarrow bS \mid cS \mid aB \mid \lambda$$

$$B \rightarrow aB \mid cS \mid bC \mid \lambda$$

$$C \rightarrow aB \mid bS \mid \lambda$$

字符串是按照从左到右的方式构造的。至多有一个变量出现在句型当中。如果有 S ，那么就不会向推导出 abc 的方向进行。当前一个终结符是 a 时，变量 B 出现。变量 C 出现仅当之前是 ab 。因此， C 规则不可能生成终结符 c 。 \square

3.3 正则文法

正则文法是上下文无关文法的重要组成子集，它在词法分析和编程语言的分析中扮演着重要的角色。正则文法是通过限制规则的右侧而获得的。在第 6 章中我们将证明正则文法可以精确地定义正则表达式定义的语言，或是有限自动机接收的语言。

定义 3.3.1 正则文法 (regular grammar) 是满足下面要求的上下文无关文法：

i) $A \rightarrow a$,

ii) $A \rightarrow aB$, 或者

iii) $A \rightarrow \lambda$,

其中, $A, B \in V$ 并且 $a \in \Sigma$ 。

[81]

正则文法中的推导具有相当好的形式。在句型中至多有一个变量。并且，如果有变量的话，变量一定是在字符串的最右侧。每个应用的规则都把终结符添加到推导的字符串中，直到应用形如 $A \rightarrow a$ 或 $A \rightarrow \lambda$ 的规则来终结整个推导。这些性质可以用正则文法 G_1

$$S \rightarrow aS \mid aA$$

$$A \rightarrow bA \mid \lambda$$

来解释。根据例 3.2.6 可以产生语言 a^+b^* 。 $aabb$ 的推导

$$S \Rightarrow aS$$

$$\Rightarrow aaA$$

$$\Rightarrow aabA$$

$$\Rightarrow aabbA$$

$$\Rightarrow aabb,$$

给出了从左到右的生成终结符的前缀的过程。推导终结于规则 $A \rightarrow \lambda$ 的使用。

使用正则文法生成的语言叫做正则 (regular) 语言。读者可以参考第 2 章介绍的正则语言家族来回顾由正则表达式描述的语言的集合。这个对同一个术语的两个不同定义是不矛盾的，因为我们证明了正则表达式和正则文法定义的是同一族的语言。

正则语言可以使用正则文法和非正则文法生成。例 3.2.6 中的文法 G_1 和 G_2 都可以生成语言 a^+b^* 。文法 G_1 不是正则的，因为规则 $S \rightarrow AB$ 不符合要求的形式。如果语言可以被某种正则文法产生，那么这种语言就是正则的。存在生成该语言的非正则文法并不影响该语言的正则性。例 3.2.9、例 3.2.10、例 3.2.11 和例 3.2.12 中的文法提供了正则文法的其他例子。

例 3.3.1 我们为一种上下文无关文法生成的语言构造一种正则文法

$$G; S \rightarrow abSA \mid \lambda$$

$$A \rightarrow Aa \mid \lambda.$$

G 的语言的是 $\lambda \cup (ab)^+a^*$ 。等价的正则文法是

[82]

$$S \rightarrow aB \mid \lambda$$

$$B \rightarrow bS \mid bA$$

$$A \rightarrow aA \mid \lambda$$

按照从左到右的方式产生字符串。 S 和 B 规则产生了集合 $(ab)^+$ 的前缀。如果字符串中有后缀 a ，则使用规则 $B \rightarrow bA$ 。 A 规则用来产生字符串的剩余部分。 \square

3.4 验证文法

前面几个小节中的文法是用来产生特定的语言的。我们已经给出直观的论据来证明文法确实产生了字符串的正确集合。但是无论论据多么有说服力,都有出现错误的可能性。因此,需要使用证明来保证一个文法能精确地产生期望的字符串。

为了证明文法 G 定义的语言和给定的语言 L 相同,就必须构造包含关系 $L \subseteq L(G)$ 和 $L(G) \subseteq L$ 。为了演示这个过程中涉及到的技术,我们将证明下面文法定义的语言

$$G: S \rightarrow AASB \mid AAB$$

$$A \rightarrow a$$

$$B \rightarrow bbb$$

就是集合 $L = \{a^{2n}b^{3n} \mid n > 0\}$ 。

如果终结字符串可以由开始符使用文法的规则推导得出,那么它就属于这种文法定义的语言。通过证明 L 中的每个字符串都可以由 G 推导,我们构造了包含关系 $\{a^{2n}b^{3n} \mid n > 0\} \subseteq L(G)$ 。因为 L 包含无穷字符串,所以我们不能构造 L 的每个字符串的推导过程。不幸的是,这恰恰就是我们需要做的。这个显而易见的困难可以使用推导框架来解决。框架用来构造 L 中的字符串的推导模式。形如 $a^{2n}b^{3n}$ ($n > 0$) 的字符串,可以使用下面的规则序列推导得出:

推 导	使用的规则
$S \xRightarrow{n-1} (AA)^{n-1} SB^{n-1}$	$S \rightarrow AASB$
$\xRightarrow{} (AA)^n B^n$	$S \rightarrow AAB$
$\xRightarrow{2n} (aa)^n B^n$	$A \rightarrow a$
$\xRightarrow{n} (aa)^n (bbb)^n = a^{2n}b^{3n}$	$B \rightarrow bbb$

其中, \Rightarrow 上的上标说明了规则的使用次数。前面的框架提供了一个“处方”,遵循它就可以产生 L 中的任何字符串的推导。

反包含 $L(G) \subseteq \{a^{2n}b^{3n} \mid n > 0\}$, 要求 G 中的每个可推导出的字符串具有集合 L 描述的形式。语言中每个字符串的推导过程中都要使用有限次规则,这就意味着可以使用归纳证明。第一个困难是要精确地给出需要证明的对象。我们希望构造 G 上所有字符串中的 a 和 b 的个数之间的关系。字符串 w 是 L 中的成员的必要条件,就是字符串 w 中 a 的个数是 b 的个数的 $2/3$ 。令 $n_x(u)$ 是字符串 u 中字符 x 的数目,那么这个关系就可以表示成 $3n_a(u) = 2n_b(u)$ 。

终结字符串中符号之间的数量关系对于 S 推导出的每个字符串未必都成立。考虑推导

$$S \Rightarrow AASB$$

$$\Rightarrow aASB.$$

由 G 推导出的字符串 $aASB$, 包含一个 a , 但没有 b 。

为了考虑推导过程中出现的中间句型,我们必须确定变量和终结符之间在所有推导步骤中总成立的那些关系。当推导出一个终结字符串时,里面不再有变量,并且这种关系就会使得产生的字符串具有所需的结构。

我们必须考察 G 中规则的变量和终结符之间的交互,从而确定它们对于推导终结字符串的作用。规则 $A \rightarrow a$ 保证每个 A 最终会被一个 a 替换。推导结果中 a 的数目包含那些已经出现在字符串中的,以及字符串中 A 的数目。 $n_a(u) + n_A(u)$ 的和代表由 u 推导出一个终结字符串中必然产生的 a 的数目。类似地,每个 B 都被字符串 bbb 替换。由 u 推导出的终结字符串中 b 的数目就是 $n_b(u) + 3n_B(u)$ 。根据上述这些观察,我们可以以此来构造条件 (i)。这个条件构造了变量和终结符之间的对应关系,并且保证这种对应关系对于推导过程的每一步都成立。

$$\text{i) } 3(n_a(u) + n_A(u)) = 2(n_b(u) + 3n_B(u))$$

可以看出,由 G 推导出的字符串 $aASB$ 满足这个条件,因为, $n_a(aASB) + n_A(aASB) = 2$ 并且 $n_b(aASB) + 3n_B(aASB) = 3$ 。

条件 (ii) 和条件 (iii) 是

$$\text{ii) } n_A(u) + n_a(u) > 1 \text{ 而且}$$

iii) 句型中的每个 a 和 A 出现在 S 之前,而 S 是在每个 b 和 B 之前。

$\{a^{2n}b^{3n} \mid n > 0\}$ 中的所有字符串包含至少两个 a 和三个 b 条件 (i) 和条件 (ii) 相结合可以得到这条性质 条件 (iii) 规定了推导字符串中字符出现的顺序 并不是所有的字符都出现在每个字符串中。使用一条规则, 由 S 推导出的字符串不可能包含终结符号。

84

确定了上述这些正确关系之后, 我们必须证明它们对于由 S 推出的每个字符串都成立 归纳的基础包括所有推导长度为 1 (S 规则) 的字符串 归纳假设保证条件对于所有推导长度不超过 n 的字符串都可以成立 并且, 归纳步骤中包括了, 证明应用一条补充规则仍能保持这种关系

有两个长度为 1 的推导 $S \Rightarrow AASB$ 和 $S \Rightarrow AAB$ 对于这些字符串中的每一个, 都有 $3(n_a(u) + n_b(u)) = 2(n_b(u) + 3n_a(u)) = 6$ 。根据观察, 条件 (ii) 和条件 (iii) 对于这两个字符串都成立

归纳假设要求 (i)、(ii) 和 (iii) 对于所有推导长度不超过 n 的字符串都成立 我们现在使用归纳假设来证明这三个性质对于应用推导长度为 $n+1$ 的所有字符串都成立

令 w 是由 S 使用长度为 $n+1$ 的推导 $S \xRightarrow{n+1} w$ 获得的字符串 为了使用归纳假设, 我们把长度为 $n+1$ 的推导写成长度为 n 的推导, 然后应用一条规则:

$$S \xRightarrow{n} u \Rightarrow w.$$

写成这种形式后, 很明显字符串 u 可以由 n 次应用推导规则获得 归纳假设断定性质 (i)、性质 (ii) 和性质 (iii) 对于 u 成立 归纳步骤需要我们证明对 u 应用一次推导规则仍保持这些性质。

对于任何句型 v , 我们令 $j(v) = 3(n_a(v) + n_b(v))$ 和 $k(v) = 2(n_b(v) + 3n_a(v))$ 根据归纳假设, $j(u) = k(u)$ 和 $j(u)/3 > 1$ 对字符串 u 的组成成分应用另一条补充规则的结果, 如下面的表格所示

规 则	$j(w)$	$k(w)$	$j(w)/3$
$S \rightarrow AASB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$S \rightarrow AAB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$A \rightarrow a$	$j(u)$	$k(u)$	$j(u)/3$
$B \rightarrow bbb$	$j(u)$	$k(u)$	$j(u)/3$

因为 $j(u) = k(u)$, 所以我们得出结论 $j(w) = k(w)$ 类似地, 根据归纳假设 $j(u)/3 > 1$ 就有 $j(w)/3 > 1$ 。每次应用每条规则的时候, 要么把 S 替换成具有适当顺序的变量序列, 要么把一个变量转化成对应的终结符, 这样获得符号的顺序就可以得到保持。

我们已经证明了对于 G 中每个可导出的字符串, 三个条件都成立 因为字符串 $w \in L(G)$ 中没有变量, 所以条件 (i) 意味着 $3n_a(w) = 2n_b(w)$ 条件 (ii) 保证 a 和 b 的存在, 而条件 (ii) 规定顺序 因此 $L(G) \subseteq \{a^{2n}b^{3n} \mid n > 0\}$ 构造完反包含关系之后, 我们得到 G 的语言是 $\{a^{2n}b^{3n} \mid n > 0\}$

正如前面证明中所解释的, 证明文法产生某种语言是个复杂的过程 当然, 上面展示的还仅仅是一个极简单的文法, 它只有几条规则 确定了正确的关系后, 归纳过程就变得很直接了 归纳证明中最具挑战的地方在于, 确定变量和终结符之间在中间句型中所必须满足的关系 如果去掉对变量的引用就可以得到具有期望结构的终结字符串, 那么这种关系就是充分的

85

正如在前面的证明中所看到的, 构造产生语言 L 的文法 G 需要满足下面两点:

i) L 中所有的字符串都可以由 G 推导出, 并且

ii) G 产生的所有字符串都属于 L 。

前者是通过提供一种可以用于产生 L 中任何字符串的推导框架来完成的 而后者则是使用归纳证明来证明每个句型都满足用以生成 L 中的字符串的条件 下面的例子进一步的解释了这些证明的步骤。

例 3.4.1 令 G 是例 3.2.10 中给出的文法

$$\begin{aligned} S &\rightarrow aS \mid bB \mid \lambda \\ B &\rightarrow aB \mid bS \mid bC \\ C &\rightarrow aC \mid \lambda. \end{aligned}$$

我们要证明 $L(G) = a^*(a^*ba^*ba^*)^*$, 它是所有 a, b 上具有偶数个 b 的字符串的集合 每个由 S 推导出的字符串都有偶数个 b , 这种说法是不正确的 推导 $S \Rightarrow bB$ 就产生了一个 b 为了推导出终结符,

每个 B 必须最后转化成一个 b 。因此, 我们得到结论, 期望的关系保证了 $n_b(u) + n_B(u)$ 是偶数。当终结字符串 w 被推导出时, $n_B(w) = 0$, 并且 $n_b(w)$ 是偶数。

我们要证明, 对于由 S 推导出的所有字符串, 都有 $n_b(u) + n_B(u)$ 是偶数的。证明是对推导的长度进行归纳。

基础步骤: 长度为 1 的推导有下面三个。

$$S \Rightarrow aS$$

$$S \Rightarrow bB$$

$$S \Rightarrow \lambda.$$

根据观察, $n_b(u) + n_B(u)$ 对于这些字符串都是偶数。

归纳假设: 假设应用上述规则 n 所得到的所有字符串 u , 都满足 $n_b(u) + n_B(u)$ 是偶数。

归纳步骤: 为了完成证明, 我们需要证明当每个 w 都是由形如 $S \xRightarrow{n+1} w$ 的推导获得时, $n_b(w) + n_B(w)$ 是偶数。关键的步骤是要重新描述推导, 从而使这个推导很容易应用归纳假设。长度为 $n+1$ 的推导可以写成 $S \xRightarrow{n} u \Rightarrow w$ 。

根据归纳假设, $n_b(u) + n_B(u)$ 是偶数。我们证明把任何规则应用到 u 都可以保持 $n_b(u) + n_B(u)$ 的奇偶性。右侧表格给出了当应用相应的规则到 u 时, 获得的 $n_b(w) + n_B(w)$ 的值。每个规则 (除了第 2 个, 它把总和增加 2) 使得 B 和 b 的个数固定不变。因此, 应用一条规则到 u 获得的字符串中 b 和 B 的个数和是偶数。因为终结字符串中不包含 B , 所以我们证明了 $L(G)$ 中的每个字符串都有偶数个 b 。

为了完成这个证明, 我们必须完成反包含关系 $L(G) \subseteq a^*(a^*ba^*ba^*)^*$ 的证明。为了证明它, 我们

要证明 $a^*(a^*ba^*ba^*)^*$ 中的每个字符串都可以由 G 推导出。 $a^*(a^*ba^*ba^*)^*$ 中的字符串具有形式

$$a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_k}ba^{n_{k+1}}, k \geq 0.$$

a^* 中的任何字符串都可以使用 $S \rightarrow aS$ 和 $S \rightarrow \lambda$ 获得。 $L(G)$ 中的所有其他字符串都可以使用下面形式的推导得到。

推 导	应用的规则
$S \xRightarrow{n_1} a^{n_1}S$	$S \rightarrow aS$
$\Rightarrow a^{n_1}bB$	$S \rightarrow bB$
$\xRightarrow{n_2} a^{n_1}ba^{n_2}B$	$B \rightarrow aB$
$\Rightarrow a^{n_1}ba^{n_2}bS$	$B \rightarrow bS$
\vdots	
$\xRightarrow{n_{2k}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}B$	$B \rightarrow aB$
$\Rightarrow a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}bC$	$B \rightarrow bC$
$\xRightarrow{n_{2k+1}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}ba^{n_{2k+1}}C$	$C \rightarrow aC$
$\Rightarrow a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}ba^{n_{2k+1}}$	$C \rightarrow \lambda$

□ [87]

例 3.4.2 已知 G 是文法

$$S \rightarrow aASB \mid \lambda$$

$$A \rightarrow ad \mid d$$

$$B \rightarrow bb.$$

我们证明 $L(G)$ 中的每个字符串中 a 和 b 的个数相等。终结字符串中 b 的个数依赖于推导的中间步骤中的 b 和 B 的个数。每个 B 产生两个 b , 而一个 A 至多产生一个 a 。我们要证明, 对于 G 的每个句型

u , 都有 $n_a(u) + n_A(u) \leq n_b(u) + 2n_B(u)$ 。已知 $j(u) = n_a(u) + n_A(u)$ 和 $k(u) = n_b(u) + 2n_B(u)$ 。

基础步骤: 有两个长度为 1 的推导。

并且, 对于这两个推导得出的字符串都有 $j(u) \leq k(u)$ 。

归纳假设: 假设对于所有由 S 推导的长度不少于 n 的字符串 u , 都有 $j(u) \leq k(u)$ 。

归纳步骤: 我们需要证明当 $S \xRightarrow{n+1} w$ 时, $j(w) \leq k(w)$ 。 w 的推导可以重写为 $S \xRightarrow{n} u \Rightarrow w$, 根据归纳假设, $j(u) \leq k(u)$ 。我们必须证明使用另外一条规则可以保持这种不等关系。每次对 j 和 k 应用规则的结果如下表所示:

规则	$j(u)$	$k(u)$
$S \Rightarrow aASB$	2	2
$S \Rightarrow \lambda$	0	0

第一条规则给不等式的两侧增加 2, 从而保持了这种不等关系。最后的规则从比较小的一侧减去 1, 从而增强了这种不等关系。对于字符串 $w \in L(G)$, 可以得到不等关系 $n_a(w) \leq n_b(w)$ 。 \square

例 3.4.3 在例 3.2.2 中, 文法

$G: S \rightarrow aSdd \mid A$

$A \rightarrow bAc \mid bc$

规则	$j(w)$	$k(w)$
$S \rightarrow aASB$	$j(u) + 2$	$k(u) + 2$
$S \rightarrow \lambda$	$j(u)$	$k(u)$
$B \rightarrow bb$	$j(u)$	$k(u)$
$A \rightarrow ad$	$j(u)$	$k(u)$
$A \rightarrow d$	$j(u) - 1$	$k(u)$

是用来构造语言 $L = \{a^n b^m c^m d^{2n} \mid n \geq 0, m > 0\}$ 的。我们构造了足以证明 $L(G) \subseteq L$ 的变量和终结符之间的关系。 S 规则和 A 规则强化了 a 和 d 以及 b 和 c 的数目之间的数字关系。在 G 的推导中, 使用规则 $S \rightarrow A$, 去掉了开始符 A 的存在保证最后会产生 b 。依据这些分析, 我们得出接下来的四个条件, 对于 G 中的每个句型 u , 都有

- i) $2n_a(u) = n_d(u)$
- ii) $n_b(u) = n_c(u)$
- iii) $n_s(u) + n_A(u) + n_b(u) > 0$
- iv) a 在 b 之前, 它们在 S 或 A 之前, 这些发生在 c 之前, 所有这些都发生在 d 之前。

等价关系保证终结符出现在正确的数字关系中。语言的描述也要求终结符以特定的顺序出现。最后的条件保证推导中每一步的顺序。 \square

3.5 最左推导和二义性

文法的语言是可以由开始符按照某种形式推导得出的终结字符串的集合。终结字符串可以由一定数目的推导产生。例如, 图 3-1 给出了使用文法规则定义的文法, 以及字符串 $ababaa$ 的四种推导。这其中的任何一种推导都足以显示字符串的语法正确性。

本章介绍的自然语言例子使用的推导都是最左推导的。对于英语的读者而言, 这是一种自然的技术, 因为最左变量是读字符串遇到的第一个变量。为了减少确定一个字符串是否属于一种文法的语言的推导步数, 我们证明语言的每个字符串都可以由最左推导得出。

定理 3.5.1 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法。字符串 w 属于 $L(G)$ 当且仅当 w 存在由 S 开始的最左推导。

证明: 显然, 若 w 存在由 S 开始的最左推导, 则有 $w \in L(G)$ 。我们必须构造等价的“仅当”关系, 即: $L(G)$ 中的每个字符串都可以最左推导出。已知

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n = w$$

是 w 在 G 中的推导, 但不必一定是最左的。上下文无关文法中, 规则应用的无关性可以用于构造 w 的最左推导。已知 w_k 是应用推导规则的最后一个句型, 但是这个推导不是最左的。如果没有这样的 k , 那么推导就是最左的, 于是就不需要证明了。我们表明记录规则的应用过程, 从而保证前 $k+1$ 条规则都是按照最左推导的原则应用的。如果需要的话, 我们可以重复这个过程 $n-k$ 次, 从而产生一个最左推导。

选择 w_k , 推导 $S \xRightarrow{k} w_k$ 是最左的。假设 A 是 w_k 的最左变量, B 是在推导的第 $k+1$ 步进行转换的变

量。 w_k 可以写成 $u_1Au_2Bu_3$, 其中 $u_i \in \Sigma^*$ 。应用规则 $B \rightarrow v$ 到 w_k , 得到

$$w_k = u_1Au_2Bu_3 \Rightarrow u_1Au_2vu_3 = w_{k+1}.$$

因为 w 是终结字符串, A 规则必须最后应用到 w_k 的最左变量上。已知第一个推导规则可以在原始推导的第 $j+1$ 个位置出现的变量 A 进行替换。那么应用规则 $A \rightarrow p$ 可以写成

$$w_j = u_1Aq \Rightarrow u_1pq = w_{j+1}.$$

在第 $k+2$ 步对 j 应用规则, 把字符串 u_2vu_3 转化成 q 。推导可以通过子推导完成

$$w_{j+1} \xRightarrow{*} w_n = w.$$

原始推导被拆分成五个子推导。所应用的前 k 个规则都是最左的, 因此它们还不是完整的。为了构造最左推导, 规则 $A \rightarrow p$ 应用到第 $k+1$ 步的最左变量。应用规则的上下文无关本质允许这种安排。第 $k+1$ 步规则应用最左推导得到的 w 可以按照下面这样的推导获得:

$$\begin{aligned} S &\xRightarrow{*} w_k = u_1Au_2Bu_3 \\ &\Rightarrow u_1pu_2Bu_3 && (\text{应用 } A \rightarrow p) \\ &\Rightarrow u_1pu_2vu_3 && (\text{应用 } B \rightarrow v) \\ &\xRightarrow{j-k-1} u_1pq = w_{j+1} && (\text{使用推导 } u_2vu_3 \xRightarrow{*} q) \\ &\xRightarrow{n-j-1} w_n && (\text{使用推导 } w_{j+1} \xRightarrow{*} w_n). \end{aligned}$$

每次重复这个程序, 推导就会变成“更多”最左。如果推导的长度是 n , 那么至多重复 n 次就产生了 w 的最左推导。

定理 3.5.1 并不保证文法的所有句型都能够由最左推导产生。仅仅是终结字符串的最左推导能够得到保证。考虑文法

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \lambda \\ B &\rightarrow bB \mid \lambda \end{aligned}$$

产生 a^*b^* , 句型 A 可以通过使用最左推导 $S \Rightarrow AB \Rightarrow A$ 获得。很容易看出没有 A 的最左推导。

类似的结果(练习 31)构造了使用最左推导来生成终结字符串的有效性。从 v 到 w 的最左推导和最右推导可以清楚地表示成 $v \xRightarrow{*}_L w$ 和 $v \xRightarrow{*}_R w$ 。

把我们的注意力放在最左推导上, 从而减少一个字符串的可能的多种推导。这种化简足以构造一个规范推导? 即, 文法的语言中的每个字符串是否都存在惟一的最左推导? 不幸的是, 回答是否。字符串 $ababaa$ 的两个最左推导如图 3-1。

具有若干种最左推导的字符串存在着二义性的表示。形式语言的二义性类似于自然语言经常遇到的二义性。句子 Jack was given a book by Hemingway 具有两种不同的结构分解。前置词短语 by Hemingway 可以修改动词 was given 或者名词 book。使用这些分解的结构可以构造语法正确的句子。

计算程序的编译使用了产生机器语言代码的分析器生成的推导。具有两种推导的程序的编译只使用一种可能的解释来产生执行代码。这样, 一个运气不好的程序员在调试的时候可能会碰上这样的情况, 根据编程语言的定义, 程序是完全正确的, 但是这个程序执行的行为却和预期的不同。为了避免出现这种情形, 保证程序员在任何情况下都可以正常调试, 计算机语言的定义应该构造得没有二义性。根据上述关于二义性的讨论, 我们给出下面的定义。

定义 3.5.2 如果存在字符串 $w \in L(G)$, 使得它在 G 中存在两种不同的最左推导, 那么上下文无关文法 G 就有二义性(ambiguous)。没有二义性的文法称作无二义性(unambiguous)。

例 3.5.1 已知产生 a^+ 的文法 G

$$S \rightarrow aS \mid Sa \mid a$$

G 是二义性的, 因为字符串 aa 有两种不同的最左推导:

$$\begin{aligned} S &\Rightarrow aS & S &\Rightarrow Sa \\ &\Rightarrow aa & &\Rightarrow aa. \end{aligned}$$

语言 a^+ 也可以由非二义性的文法产生

$$S \rightarrow aS \mid a.$$

如果文法是正则,那么它就具有下面的性质:所有的字符串都可以从左到右产生。变量 S 一直是字符串的最右符号直到使用规则 $S \rightarrow a$ 使得递归停止为止。 \square

前面的例子显示二义性是文法的性质,而不是语言的性质。当文法显示是二义性的时候,通常可以构造与之等价的无二义性的文法。但这并不是总能够成功的。确实存在一些不能用无二义性文法生成的上下文无关文法。此种语言的这种性质叫做固有二义性 (inherently ambiguous)。大多数编程语言的语法,需要使用无二义性的推导,所以要求严格避免固有二义性。

例 3.5.2 已知 G 是生成语言 b^*ab^* 的文法

$$S \rightarrow bS \mid Sb \mid a.$$

最左推导

$$\begin{array}{ll} S \Rightarrow bS & S \Rightarrow Sb \\ \Rightarrow bSb & \Rightarrow bSb \\ \Rightarrow bab & \Rightarrow bab \end{array}$$

表明 G 是具有二义性的。这种文法具有能够以任何顺序生成 b 的能力,因此必须去掉,从而获得无二义性的文法。 $L(G)$ 是使用下面这种无二义性的文法生成的语言:

$$\begin{array}{ll} G_1: S \rightarrow bS \mid aA & G_2: S \rightarrow bS \mid A \\ A \rightarrow bA \mid \lambda & A \rightarrow Ab \mid a. \end{array}$$

在 G_1 中,最左推导中应用规则的顺序是完全由推导出的字符串所决定的。字符串 $b^n ab^m$ 的惟一最左推导具有下面的形式:

$$\begin{array}{l} S \xRightarrow{a} b^n S \\ \Rightarrow b^n aA \\ \xRightarrow{m} b^n ab^m A \\ \Rightarrow b^n ab^m. \end{array}$$

G_2 的推导初始生成 b ,后面跟着作为尾部的 b ,最后是 a 。 \square

如果最左推导的每一步,只有惟一的一种规则可以应用才能生成期望的字符串,那么这种文法是无二义性的。但是这并不意味着当时只有一种可以应用的规则,而是应用其他的规则可能会导致无法生成期望的字符串。

考虑使用例 3.5.2 的文法 G_2 来构造字符串 $bbabbb$ 的最左推导的可能性。有两条 S 规则可以用于初始化推导过程。首先使用规则 $S \rightarrow A$,那么推导过程将产生首字母为 a 的字符串。因此, $bbabbb$ 的推导过程必须首先使用规则 $S \rightarrow bS$ 。第二个 b 是由同一条规则产生的。在此基础上,推导过程继续使用 $S \rightarrow A$,然后应用 $S \rightarrow bS$,从而生成前缀 bbb 。后缀 bb 是使用两次推导规则 $A \rightarrow Ab$,然后应用推导规则 $A \rightarrow a$ 推导生成的。因为终结字符串本身确定了使用的规则的准确顺序,所以文法是无二义性的。

例 3.5.3 例 3.2.4 中产生语言 $L = \{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$ 的文法是有二义性的。字符串 $aabbbb$ 可以由下面的推导得到:

$$\begin{array}{ll} S \Rightarrow aSb & S \Rightarrow aSbb \\ \Rightarrow aaSbbb & \Rightarrow aaSbbb \\ \Rightarrow aabbbb & \Rightarrow aabbbb. \end{array}$$

无二义性地生成 L 中的字符串的这种策略是,首先产生 a ,以及一个相匹配的 b ,然后生成一个 a 和两个 b 。按照这种方式生成 L 中字符串的无二义性文法是

$$\begin{array}{l} S \rightarrow aSb \mid A \mid \lambda \\ A \rightarrow aAbb \mid abb. \end{array} \quad \square$$

推导树描述了推导中变量的转换。在最左(最右)推导和推导树之间存在一个自然的一一对应。定义 3.1.4 概述了直接源于最左推导的推导树的构造过程。相反,一个字符串 w 的惟一最左推导可以从一个边界为 w 的推导树中抽取出来。因为这种对应关系,二义性经常由使用推导树的术语来定义。如果 $L(G)$ 中的字符串位于两棵不同的推导树的边界,那么文法 G 是二义性的。图 3-3 表明,图 3-1 中给出的字符串 $ababaa$ 的两个最左推导产生了不同的推导树。

3.6 上下文无关文法和编程语言定义

在前面的小节中,我们使用上下文无关文法,依靠仅有几个元素的字母表和几条规则来生成“玩具”语言。这些例子都显示了上下文无关规则产生满足特定语法需求的字符串的能力。编程语言具有很大的字母表和更复杂的语法,并增加了定义语言需要的规则的复杂性和数目。高级语言的第一个形式化描述是 John Backus [1959] 和 Peter Naur [1963] 提出的 ALGOL 60 语言。Backus 和 Naur 使用的这个系统称作巴克斯-瑙尔范式 (Backus-Naur form 或者 BNF)。编程语言 Java 的规约就是用 BNF 给出的,它用来解释编程语言语法定义的原则。附录 IV 给出了 Java 的完整形式化定义。

93

语言的 BNF 描述就是上下文无关文法。它们唯一的区别是用来定义规则的表示法不同。我们将使用上下文无关表示给出规则,当然除了一点例外。变量或终结符后面的下角标 *opt* 表明它是可选的。这种表示减少了需要写出的规则的数目,但是带有可选构成成分的规则可以很容易地转化成等价的上下文无关规则。例如, $A \rightarrow B_{opt}$ 和 $A \rightarrow B_{opt} C$ 可以分别用规则 $A \rightarrow B \mid \lambda$ 和 $A \rightarrow BC \mid C$ 代替。

Java 规则中使用的表示习惯与本章开始时所给出的例子中的自然语言是一样的。变量的名字指的是它们产生的语言的构成成分,并且把这个名字放在 `< >` 中。Java 关键词使用粗体,其他的终结符号使用字符串,中间用空格表示。

编程语言的设计和复杂程序的设计类似,是使用模块化来分别构造文法的子集,从而得到简化了的设计过程。在构造小规则集中使用的技巧,也可以用于设计具有复杂语法的大型语言的文法。这些技巧包括使用规则保证成员的存在,成员的相对位置,或是使用递归生成序列,或生成成对的括号。

为了解释设计语言的原则,我们考察定义 Java 中常量、标识符和算术表达式的规则。常量和具有固定类型和值的字符串常常被用来初始化变量,从而为重复的语句设置界限,储存标准信息用以输出。变量 *Literal* 的规则定义了 Java 常量的类型。Java 常量以及产生它们的变量如下表所示:

常 量	变 量	例 子
Boolean	<code>< BooleanLiteral ></code>	true, false
Character	<code>< CharacterLiteral ></code>	'a', '\n' (换行符), '\u003cpi>',
String	<code>< StringLiteral ></code>	" " (空串), "This is a nonempty string"
Integer	<code>< IntegerLiteral ></code>	0, 356, 1234L (长整型), 077 (八进制), 0x1ab2 (十六进制)
Floating point	<code>< FloatingPointLiteral ></code>	2., .2, 2.0, 12.34, 2e3, 6.2e-5
Null	<code>< NullLiteral ></code>	null

每个浮点常量都有 f, F, d 或 D 来作为后缀表示它的准确度。完整的 Java 常量集合定义参见附录 IV 中的规则 143–167。

94

我们进一步考虑定义浮点常量的规则,因为它们含有最有趣的语法变量。四个 `< FloatingPointLiteral >` 规则指定了浮点常量的通用形式。

$$\begin{aligned} \langle \text{FloatingPointLiteral} \rangle \rightarrow & \langle \text{Digits} \rangle . \langle \text{Digits} \rangle_{opt} \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle_{opt} \mid \\ & . \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle_{opt} \mid \\ & \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle \langle \text{FloatTypeSuffix} \rangle_{opt} \mid \\ & \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle \end{aligned}$$

变量 *Digits*、*ExponentPart* 和 *FloatTypeSuffix* 产生了组成常量的构成成分。变量 *Digits* 使用递归产生了数字字符串。非递归规则保证了至少一个数字的存在。

$$\begin{aligned} \langle \text{Digits} \rangle \rightarrow & \langle \text{Digit} \rangle \mid \langle \text{Digits} \rangle \langle \text{Digit} \rangle \\ \langle \text{Digit} \rangle \rightarrow & 0 \mid \langle \text{NonZeroDigit} \rangle \\ \langle \text{NonZeroDigit} \rangle \rightarrow & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \langle \text{ExponentPart} \rangle \rightarrow & \langle \text{ExponentIndicator} \rangle \langle \text{SignedInteger} \rangle \end{aligned}$$

$$\begin{aligned}\langle \text{ExponentIndicator} \rangle &\rightarrow \mathbf{e} \mid \mathbf{E} \\ \langle \text{SignedInteger} \rangle &\rightarrow \langle \text{Sign} \rangle_{\text{opt}} \langle \text{Digits} \rangle \\ \langle \text{Sign} \rangle &\rightarrow + \mid - \\ \langle \text{FloatTypeSuffix} \rangle &\rightarrow \mathbf{f} \mid \mathbf{F} \mid \mathbf{d} \mid \mathbf{D}\end{aligned}$$

规则 $\text{SignedInteger} \rightarrow \text{Sign} \cdot_{\text{opt}} \text{Digits}$ 规则中的 *opt* 表示一个带符号的整数可以由 + 或 - 开始, 但是符号不是必须的。

第一条 $\langle \text{FloatingPointLiteral} \rangle$ 规则产生了形如 1., 1.1, 1.1e, 1.e, 1.1ef, 1.f, 1.1.f 和 1.ef 的常量。具有数字和十进制点的领头字符串是必须的, 所有其他的构成成分是可选的。第二条规则产生了以十进制点开始的常数, 而最后两条规则定义了没有十进制点的浮点常数。

标识符用来定义变量、类型和方法等的名字。标识符使用下面的规则定义:

$$\begin{aligned}\langle \text{Identifier} \rangle &\rightarrow \langle \text{IdentifierChars} \rangle \\ \langle \text{IdentifierChars} \rangle &\rightarrow \langle \text{JavaLetter} \rangle \mid \langle \text{JavaLetter} \rangle \langle \text{JavaLetterOrDigit} \rangle\end{aligned}$$

其中, Java 字母包括字母从 A 到 Z, 从 a 到 z, 下划线 _ 和美国符号 \$, 以及其他用 Unicode 编码的字符。

Java 中的语句的定义开始于变量 $\langle \text{Statement} \rangle$:

$$\begin{aligned}\langle \text{Statement} \rangle &\rightarrow \langle \text{StatementWithoutTrailing Substatement} \rangle \mid \langle \text{LabeledStatement} \rangle \mid \\ &\quad \langle \text{IfThenStatement} \rangle \mid \langle \text{IfThenElseStatement} \rangle \mid \\ &\quad \langle \text{WhileStatement} \rangle \mid \langle \text{ForStatement} \rangle.\end{aligned}$$

没有子语句的语句包括块, 和 do 语句与 switch 语句。附录 IV 中的规则 73 - 75 给出语句的整个集合。与常量的规则类似, 语句规则定义了语句的高层结构。例如, if-then 和 do 语句是由下面来定义的。

$$\begin{aligned}\langle \text{IfThenStatement} \rangle &\rightarrow \text{if}(\langle \text{Expression} \rangle) \langle \text{Statement} \rangle \\ \langle \text{DoStatement} \rangle &\rightarrow \text{do} \langle \text{Statement} \rangle \text{while}(\langle \text{Expression} \rangle)\end{aligned}$$

变量 Statement 出现在前面规则的右侧, 从而生成了 if-then 语句中的条件语句之后, 和 do 循环体中的执行的语句。

表达式是评估是数字计算和检查 if-then、do、while 和 switch 语句的关键。表达式的语法是使用附录 IV 中的规则 118 - 142 来定义。语法很复杂, 因为 Java 具有数学表达式和布尔表达式, 它们可能使用后缀操作符、前缀操作符或是中缀操作符。除了描述这些单独的规则外, 我们还将看看出现在简单数学赋值的推导过程中存在的几个子推导。

第一步是把变量 $\langle \text{Expression} \rangle$ 转化成赋值句。

$$\begin{aligned}\langle \text{Expression} \rangle &\Rightarrow \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Assignment} \rangle \\ &\Rightarrow \langle \text{LeftHandSide} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{ExpressionName} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle = \langle \text{AssignmentExpression} \rangle.\end{aligned}$$

下一步就是由 $\langle \text{AssignmentExpression} \rangle$ 推导出 $\langle \text{AdditiveExpression} \rangle$ 。

$$\begin{aligned}\langle \text{AssignmentExpression} \rangle &\Rightarrow \langle \text{ConditionalExpression} \rangle \\ &\Rightarrow \langle \text{ConditionalOrExpression} \rangle \\ &\Rightarrow \langle \text{ConditionalAndExpression} \rangle \\ &\Rightarrow \langle \text{InclusiveOrExpression} \rangle \\ &\Rightarrow \langle \text{ExclusionOrExpression} \rangle \\ &\Rightarrow \langle \text{AndExpression} \rangle \\ &\Rightarrow \langle \text{EqualityExpression} \rangle \\ &\Rightarrow \langle \text{RelationalExpression} \rangle\end{aligned}$$

$$\Rightarrow \langle \text{ShiftExpression} \rangle$$

$$\Rightarrow \langle \text{AdditiveExpression} \rangle.$$

96

由 $\langle \text{AdditiveExpression} \rangle$ 开始的推导产生形式正确的带有加法操作符、乘法操作符和括号的表达式。比如,

$$\begin{aligned} \langle \text{AdditiveExpression} \rangle &\Rightarrow \langle \text{AdditiveExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{MultiplicativeExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{UnaryExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle + \langle \text{MultiplicativeExpression} \rangle * \langle \text{MultiplicativeExpression} \rangle \end{aligned}$$

开始了这样一个推导。由 $\langle \text{UnaryExpression} \rangle$ 开始的推导能够产生常数、变量或者是获得嵌套括号的 ($\langle \text{Expression} \rangle$)。

这些定义了标识符、常量和表达式的规则显示了大型语言的设计是如何分解成语言中经常出现的子集的规则。结构变量 $\langle \text{Identifier} \rangle$ 、 $\langle \text{Literal} \rangle$ 和 $\langle \text{Expression} \rangle$ 为高层规则构造了模块。

文法的开始符是 $\langle \text{CompilationUnit} \rangle$, Java 程序的推导源于规则

$$\langle \text{CompilationUnit} \rangle \rightarrow \langle \text{PackageDeclaration} \rangle_{opt} \langle \text{ImportDeclarations} \rangle_{opt} \langle \text{TypeDeclarations} \rangle_{opt}$$

由这条规则推导出的终结符构成的字符串是语法正确的 Java 程序。

3.7 练习

1. 已知 G 是文法

$$\begin{aligned} S &\rightarrow abSc \mid A \\ A &\rightarrow cAd \mid cd. \end{aligned}$$

- 给出 $ababccddcc$ 的推导。
- 给 (a) 中推导过程构造的对应的推导树。
- 使用集合表示来定义 $L(G)$ 。

2. 已知 G 是文法

$$\begin{aligned} S &\rightarrow ASB \mid \lambda \\ A &\rightarrow aAb \mid \lambda \\ B &\rightarrow bBa \mid ba. \end{aligned}$$

- 给出 $aabbba$ 的最左推导。
- 给出 $abaabbbabbaa$ 的最右推导。
- 构造 (a) 和 (b) 中推导对应的推导树。
- 使用集合表示法来定义 $L(G)$ 。

97

3. 已知 G 是文法

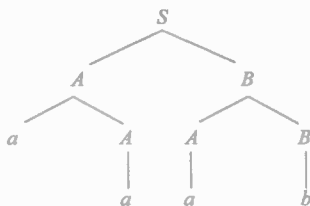
$$\begin{aligned} S &\rightarrow SAB \mid \lambda \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid \lambda. \end{aligned}$$

- 给出 $abbaab$ 的最左推导。
- 给出 aa 的两种最左推导。
- 构造 (b) 中推导对应的推导树。
- 给出 $L(G)$ 的正则表达式。

4. 已知 DT 是推导树

- 给出产生树 DT 的一个最左推导。
- 给出产生树 DT 的一个最右推导。
- 有多少个产生 DT 的不同的推导。

5. 给出图 3-3 中每个推导树的最左和最右推导。



6. 对于下面的每种上下文无关文法, 使用集合表示来定义这些文法产生的语言。

- a) $S \rightarrow aaSB \mid \lambda$
 $B \rightarrow bB \mid b$
- b) $S \rightarrow aSbb \mid A$
 $A \rightarrow cA \mid c$
- c) $S \rightarrow abSdc \mid A$
 $A \rightarrow cdAba \mid \lambda$
- d) $S \rightarrow aSb \mid A$
 $A \rightarrow cAd \mid cBd$
 $B \rightarrow aBb \mid ab$
- e) $S \rightarrow aSB \mid aB$
 $B \rightarrow bb \mid b$

7. 构造 $\{a, b, c\}$ 上语言 $\{a^n b^{2n} c^m \mid n, m > 0\}$ 的文法。

8. 构造 $\{a, b, c\}$ 上语言 $\{a^n b^m c^{2n+m} \mid n, m > 0\}$ 的文法。

98. 9. 构造 $\{a, b, c\}$ 上语言 $\{a^n b^m c^i \mid 0 \leq n+m \leq i\}$ 的文法。

10. 构造 $\{a, b\}$ 上语言 $\{a^n b^n \mid 0 \leq n \leq m \leq 3n\}$ 的文法。

11. 构造 $\{a, b\}$ 上语言 $\{a^m b^i a^n \mid i = m+n\}$ 的文法。

12. 构造 $\{a, b\}$ 上包含相同数目 a 和 b 的字符串的语言的文法。

13. 构造 $\{a, b\}$ 上长度为奇数, 并且字符串的首位和中间位置具有相同字符的字符串构成的语言的文法。

14. 对于下面每种正则文法, 给出它们产生的语言对应的正则表达式。

- a) $S \rightarrow aA$
 $A \rightarrow aA \mid bA \mid b$
- b) $S \rightarrow aA$
 $A \rightarrow aA \mid bB$
 $B \rightarrow bB \mid \lambda$
- c) $S \rightarrow aS \mid bA$
 $A \rightarrow bB$
 $B \rightarrow aB \mid \lambda$
- d) $S \rightarrow aS \mid bA \mid \lambda$
 $A \rightarrow aA \mid bS$

从练习 15 到练习 25, 分别给出产生相应语言的正则文法。

15. $\{a, b, c\}$ 的字符串, 其中, 所有 a 发生在 b 的前面, 这些都出现在 c 的前面 (可以存在没有 a, b , 或者没有 c 的情况)。
16. $\{a, b\}$ 上包含子串 aa 和 bb 的字符串的集合。
17. $\{a, b\}$ 上子串 aa 至少出现两次的字符串的集合 (提示: 注意字符串 aaa)。
18. $\{a, b\}$ 上包含子串 ab 和 ba 的字符串的集合。
19. $\{a, b\}$ 上 a 的个数可以被 3 整除的字符串的集合。
20. $\{a, b\}$ 上字符串, a 要么紧跟在 b 的前面, 要么紧跟在 b 的后面。例如, $baab, aba$ 和 b 。
21. $\{a, b\}$ 上不包含子串 aba 的字符串的集合。
22. $\{a, b\}$ 上子串 aa 只出现一次的字符串的集合。
23. $\{a, b\}$ 上只包含两个 b 的字符串的集合。
24. $\{a, b, c\}$ 上子串 ab 出现的次数是奇数的字符串的集合。
25. $\{a, b\}$ 上具有偶数个 a , 或者有奇数个 b 的字符串的集合。
26. 图 3-1 中的文法产生了 $(b^* ab^* ab^*)^+$, 它是所有具有正偶数个 a 的字符串的集合 (证明这个结论)
27. 证明例 3.2.2 中给出的文法确实产生了期望的语言。
28. 已知 G 是文法

$$\begin{aligned} S &\rightarrow aSb \mid B \\ B &\rightarrow bB \mid b. \end{aligned}$$

99. 证明 $L(G) = \{a^n b^m \mid 0 \leq n < m\}$ 。

29. 已知 G 是文法

$$\begin{aligned} S &\rightarrow aSaa \mid B \\ B &\rightarrow bbBdd \mid C \\ C &\rightarrow bd. \end{aligned}$$

a) 什么是 $L(G)$?

b) 证明 $L(G)$ 是 (a) 给出的集合。

*30. 已知 G 是文法

$$S \rightarrow aSbS \mid aS \mid \lambda.$$

证明 $L(G)$ 的字符串的每个前缀 a 的个数不少于 b 。

31. 已知 G 是上下文无关文法, 并且 $w \in L(G)$ 。证明 G 中的 w 存在最右推导。

32. 已知 G 是文法

$$S \rightarrow aS \mid Sb \mid ab.$$

a) 给出 $L(G)$ 的正则表达式。

b) 构造字符串 $aabb$ 的两种最左推导。

c) 构造 (b) 中推导过程对应的推导树。

d) 构造等价于 G 的无二义性文法。

33. 对于下面每种文法, 给出相应文法的语言的正则表达式或集合论定义。证明文法是二义性的, 并给出相应的无二义性文法。

a) $S \rightarrow aaS \mid aaaaaS \mid \lambda$

b) $S \rightarrow aSA \mid \lambda$

$A \rightarrow bA \mid \lambda$

c) $S \rightarrow aSb \mid aAb$

d) $S \rightarrow AaSbB \mid \lambda$

$A \rightarrow cAd \mid B$

$A \rightarrow aA \mid a$

$B \rightarrow aBb \mid \lambda$

$B \rightarrow bB \mid \lambda$

* e) $S \rightarrow A \mid B$

$A \rightarrow abA \mid \lambda$

$B \rightarrow aBb \mid \lambda$

34. 已知文法

$$S \rightarrow aA \mid \lambda$$

$$A \rightarrow aA \mid bB$$

$$B \rightarrow bB \mid b.$$

a) 给出 $L(G)$ 的正则表达式。

b) 证明 G 是无二义性的。

35. 已知文法 G 是

$$S \rightarrow aS \mid aA \mid a$$

$$A \rightarrow aAb \mid ab.$$

a) 给出 $L(G)$ 的集合论定义。

b) 证明 G 是无二义性的。

36. 已知文法 G 是

$$S \rightarrow aS \mid bA \mid \lambda$$

$$A \rightarrow bA \mid aS \mid \lambda.$$

给出 $L(G)$ 的正则表达式。 G 是二义性的吗? 如果是, 给出产生 $L(G)$ 的无二义性文法。如果不是, 请证明。

37. 为语言 $L_1 = \{a^n b^m c^m \mid n, m > 0\}$ 和 $L_2 = \{a^n b^m c^m \mid n, m > 0\}$ 构造无二义性文法。构造产生 $L_1 \cup L_2$ 的文法 G 。证明 G 是二义性的。这是具有固有二义性的语言的一个实例。给出直观的解释, 为什么产生 $L_1 \cup L_2$ 的每个文法都是有二义性的。

38. 使用附录 IV 中的 Java 定义从变量 $\langle Literal \rangle$ 来构造字符串 $1.3e2$ 的推导。

*39. 已知 G_1 和 G_2 是下面的文法

$$G_1: S \rightarrow aABb$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

$$G_2: S \rightarrow AABB$$

$$A \rightarrow AA \mid a$$

$$B \rightarrow BB \mid b.$$

a) 对于每个变量 X , 证明 G_1 的每个 X 的右侧都可以使用规则 G_2 由对应的变量 X 推导出。使用这点来证明 $L(G_1) \subseteq L(G_2)$ 。

b) 证明 $L(G_1) = L(G_2)$ 。

40. 右线性文法 (right-linear grammar) 是上下文无关文法, 它的每条规则都至少有下面的形式之一。

i) $A \rightarrow w$, 或者

ii) $A \rightarrow wB$,

其中, $w \in \Sigma^*$ 。证明语言 L 是由右线性文法产生的, 当且仅当它可以由一种正则文法生成。

41. 试着构造产生语言 $\{a^n b^n \mid n \geq 0\}$ 的正则文法。解释为什么你的尝试无法成功。

42. 试着构造产生语言 $\{a^n b^n c^n \mid n \geq 0\}$ 的上下文无关文法。解释为什么你的尝试无法成功。

参考文献注释

上下文无关文法是 Chomsky [1956], [1959] 提出的。巴克斯-瑙尔范式是由 Backus [1959] 提出的。这些形式都曾用来定义变成语言 ALGOL, 参见 Naur [1963]。附录 IV 中给出了 Java 的 BNF 定义。Ginsburg 和 Rice [1962] 记录了 BNF 定义的语言和上下文无关语言的等价性。

Floyd [1962]、Cantor [1962] 以及 Chomsky 和 Schutzenberger [1963] 都考查了二义性这种性质。练习 37 中的语言是固有二义性的证明可参考 Harrison [1978]。Ginsburg 和 Ullian [1966a, 1966b] 构造了二义性和固有二义性的语言的封闭性质。

第4章 上下文无关文法范式

上下文无关文法的定义允许规则右端的形式无限灵活。这种灵活性对于设计文法非常有益,但是由于上下文无关文法缺乏结构性,因此使得我们很难在文法、推导和语言之间建立一般的联系。上下文无关文法范式在文法的基础上加入了结构上的限制,使得上下文无关文法和语言的分析变得更容易。范式有两个特点:

- i) 满足范式的文法就可以生成整个上下文无关语言。
- ii) 存在一个算法,可以将任意上下文无关文法转化为与其等价的上下文无关文法范式。

本章我们介绍了两种重要的上下文无关文法范式,乔姆斯基范式和格立巴赫范式。我们也给出了任意上下文无关文法转换成它们的方法。这些转换过程包括规则的修改以及增删等操作,每个操作都保持最初文法生成的语言不变。

范式确保了文法的推导满足一定的条件。乔姆斯基范式文法的推导树是一叉树。在第7章,我们会利用二叉树的深度和叶子数的关系来确保一个上下文无关语言中存在重复的模式。我们也会利用乔姆斯基范式文法中推导的特性来设计高效的算法,并用于判断一个字符串是否属于某个文法描述的语言。 [103]

格立巴赫范式文法的推导采用从左到右的方式来构造字符串。每条规则为推导字符串添加一个终结符。在第7章中我们会用格立巴赫范式来为上下文无关语言建立基于机器的特征。

4.1 文法转换

一个文法到范式的转换是由一系列规则的增、删、改构成。每一次修改都不改变文法对应的语言。每一步的目标都是产生满足某些性质的规则。这些转换的顺序是为了确保接下来的每一步转换,都仍然保持前面步骤所产生的性质。

第一个转换非常简单。我们的目标是限制文法初始符号对推导过程初始化的影响。如果文法的开始符是递归变量,那么 $S \Rightarrow uSv$ 形式的推导使得文法开始符始终出现在每一次推导的句型中。对于每个文法 G ,我们可以找到一个等价文法 G' ,在 G' 中,文法开始符不是递归的。 G 和 G' 的文法的开始符并不一定是一样的。虽然转换过程很简单,但是这个转换过程显示出,必须存在步骤来证明转换并没有修改原文法的语言。

引理 4.1.1 设 $G = (V, \Sigma, P, S)$ 是一个上下文无关文法,存在一个文法 G' 满足

- i) $L(G) = L(G')$ 。
- ii) G' 的开始符不是递归变量。

证明: 如果 G 的开始符 S 并没有出现在 G 的任意一个推导规则的右侧,那么 $G = G'$ 。如果 S 是一个递归变量,那么使它产生递归的规则必须被去掉。我们可以采用“回退”的方法。构建一个新文法 $G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S, S \rightarrow u\}, S')$ 。我们用 S' 来代替 S ,在规则集中加入规则 $S' \rightarrow S$ 。这两种文法产生的语言完全相同。因为任意一个可以从 G 推导出来的字符串 u ,即 $S \xRightarrow{G} u$,都可以从推导 $S' \Rightarrow S \xRightarrow{G} u$ 中获得。而且,新添加的规则唯一的作用是初始化 G' 的推导。因此 G' 中推导出来的字符串也可以从 G 中推导出来。 [104]

例 4.1.1 G 的开始符

$G: S \rightarrow aS \mid AB \mid AC$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow bB \mid bS$
 $C \rightarrow cC \mid \lambda$

$G': S' \rightarrow S$
 $S \rightarrow aS \mid AB \mid AC$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow bB \mid bS$
 $C \rightarrow cC \mid \lambda$

是递归的,我们用引理 4.1.1 中介绍的方法构建了等价文法 G' 。 G' 的开始符是 S' , S' 不是递归的。 G' 中变量 S 仍然是递归的,但是它已经不是开始变量了。□

文法转换到范式的过程包含规则的增加和删除。每一步的变化都不应该修改文法产生的语言。引理 4.1.2 介绍了如何保证增加规则也不会改变语言本身。引理 4.1.3 介绍了如何删除规则。删除一条规则的同时,显然必须增加新的规则,否则语言就会改变。

引理 4.1.2 设 $G = (V, \Sigma, P, S)$ 是一个上下文无关文法,如果 $A \xRightarrow{G} w$, 那么文法 $G' = (V, \Sigma, P \cup \{A \rightarrow w\}, S)$ 和 G 等价。

证明: 很明显 $L(G) \subseteq L(G')$, 这是因为 G 中每一条规则都在 G' 中。而 $L(G') \subseteq L(G)$ 是因为 G' 中规则 $A \rightarrow w$ 导出的串也可以由 $A \xRightarrow{G} w$ 导出。■

引理 4.1.3 设 $G = (V, \Sigma, P, S)$ 是一个上下文无关文法, $A \rightarrow uBv$ 是 P 中一条规则。而且 $B \rightarrow w_1 | w_2 | \dots | w_n$ 是 P 中的规则。文法 $G' = (V, \Sigma, P', S)$, 其中

$$P' = (P - \{A \rightarrow uBv\}) \cup \{A \rightarrow uw_1v | uw_2v | \dots | uw_nv\}$$

和 G 等价。

证明: 既然每条规则 $A \rightarrow uw_iv$ 都是可以从 G 推导出来的, 那么根据引理 4.1.2, $L(G') \subseteq L(G)$ 。

另一方面, G 中使用规则 $A \rightarrow uBv$ 推导出来的终结符串也可以从 G' 中推导出来。使用这条规则推导出的终结符串的最右推导过程如下:

$$S \Rightarrow pAq \Rightarrow puBvq \Rightarrow pxBvq \Rightarrow pxw_ivq \Rightarrow w,$$

这里 $u \Rightarrow x$ 将 u 转为一个终结符串。这个串可以在 G' 使用规则 $A \rightarrow uw_iv$ 产生:

$$S \Rightarrow pAq \Rightarrow puw_ivq \Rightarrow pxw_ivq \Rightarrow w.$$

4.2 消去 λ 规则

在终结符串的推导过程中,中间串可以包含那些不会产生终结符的变量。这些变量可以用 λ 规则去掉。我们可以从以下文法中推导出串 $aaaa$:

$$S \rightarrow SaB | aB$$

$$B \rightarrow bB | \lambda.$$

从这个文法中产生的语言是 $(ab^*)^+$ 。 $aaaa$ 是根据第二条规则生成的。每产生一个 B , 就可以用 $B \rightarrow \lambda$ 消去。

$$S \Rightarrow SaB$$

$$\Rightarrow SaBaB$$

$$\Rightarrow SaBaBaB$$

$$\Rightarrow aBaBaBaB$$

$$\Rightarrow aaBaBaB$$

$$\Rightarrow aaaBaB$$

$$\Rightarrow aaaaB$$

$$\Rightarrow aaaa.$$

我们下一个转换过程的目标就是确保句子中的每一个变量都产生终结符串中的终结符。在上面的例子中,没有一个 B 产生终结符。一个更加有效的方法是避免产生那些会被 λ 规则消去的变量。

语言 $(ab^*)^+$ 可以由这个文法产生:

$$S \rightarrow SaB | Sa | aB | a$$

$$B \rightarrow bB | b$$

这个文法没有 λ -规则。串 $aaaa$ 的推导过程如下:

$$S \Rightarrow Sa$$

$$\Rightarrow Saa$$

$$\Rightarrow Saaa$$

$$\Rightarrow aaaa,$$

这个推导过程只使用了前面推导规则的一半,这种效率的获得是以文法规则数目的增加为代价的。

推理过程中 λ 规则 $B \rightarrow \lambda$ 不仅仅影响变量 B 。看看这个文法:

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow aA \mid B \\ B &\rightarrow bB \mid \lambda. \end{aligned}$$

这个文法产生语言 a^*b^* 。变量 A 出现在字符串 ab 的推导过程中,但是接下来使用了规则 $A \rightarrow B$,没有产生终结符。

$$\begin{aligned} S &\Rightarrow aAb \\ &\Rightarrow aBb \\ &\Rightarrow ab. \end{aligned}$$

只要一个变量可以产生空串,就像 A ,那么它就可能不产生终结符。我们将这种可以产生空串的变量叫做可空变量(nullable)。如果句子中包含一个可空变量,那么句子的长度可能会减少。

下面介绍消去 λ -规则的方法。消去过程包含三步:

1. 确定可空变量。
2. 去掉出现了可空变量的规则。
3. 去掉 λ -规则。

如果一个语法没有可空变量,那么推导中出现的每一个变量都会产生终结符。这样使用规则就不会缩短句子的长度。这样的文法称为非收缩性(noncontracting)的。

消去过程的第一步是确定可空变量集。可以使用算法 4.2.1 递归地从 λ -规则中获得可空变量集。这个算法使用了两个集合: NULL 集合是可空变量集, PREV 集合是上次迭代过程中得到的可空变量集,它是触发算法的停止条件。

107|

算法 4.2.1

可空变量集的构造

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

1. NULL: = $\{A \mid A \rightarrow \lambda \in P\}$
2. repeat
 - 2.1. PREV: = NULL
 - 2.2. for 任意 $A \in V$ do

if 存在一个 A 规则 $A \rightarrow w$ 且 $w \in \text{PREV}^*$, then

NULL: = NULL $\cup \{A\}$
- until NULL = PREV

NULL 集合首先由那些直接产生空串的变量组成。如果某个规则中左边是变量 A ,右边由 NULL 集合中的变量构成,那么 A 就被加到 NULL 集合中。如果算法无法找到新的变量 A ,那么算法就停止了。既然变量的数目是有限的,那么算法就必然会停止。可空变量的定义依赖于推导的表示,它是递归的。因此,可以用归纳法来证明 NULL 集合中包含的都是计算结束时 G 中的可空变量。

引理 4.2.2 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。算法 4.2.1 产生的是 G 的可空变量集合。

证明: 用归纳法证明 NULL 中的每个变量都会产生空串。如果 A 在第一步被加入到 NULL,那么 G 包含规则 $A \rightarrow \lambda$,因此 A 可以推导出空串。

假设经过 n 次迭代之后 NULL 中包含的变量都是可空的。我们必须证明在第 $n+1$ 次迭代过程中新加入的变量也是可空的。如果 A 是这样一个变量,那么就存在这样一个规则:

$$A \rightarrow A_1 A_2 \cdots A_k$$

每个 A_i 都在 PREV 中。根据归纳假设, $A_i \Rightarrow \lambda$ $i=1,2,\dots,k$ 。根据这些推导,有

108

$$\begin{aligned}
 A &\Rightarrow A_1 A_2 \cdots A_n \\
 &\Rightarrow A_1 \cdots A_n \\
 &\Rightarrow A_1 \cdots A_n \\
 &\vdots \\
 &\Rightarrow A_i \\
 &\Rightarrow \lambda,
 \end{aligned}$$

可见 A 是可空变量。现在我们来证明所有的可空变量都在 NULL 中。如果 n 是从 A 推导出空串的最小推导次数, 那么 A 就在第 n 次迭代或者第 n 步之前被加入到 NULL 中。使用归纳法对 A 推导出空串的推导次数进行归纳。

如果 $A \Rightarrow \lambda$, 那么 A 在第一步被加入到 NULL。假设所有最小推导次数小于等于 n 的变量都在第 n 次或者第 n 步之前加入到 NULL 中。设 A 是最小推导次数为 $n+1$ 的可空变量。推导过程为:

$$\begin{aligned}
 A &\Rightarrow A_1 A_2 \cdots A_i \\
 &\Rightarrow \lambda.
 \end{aligned}$$

每个变量 A_i 都是最小推导次数小于等于 n 的可空变量。根据归纳假设, 每个 A_i 都在前 $n+1$ 次迭代被加入到 NULL。设 $m \leq n$, 假设所有的 A_i 都在第 m 次迭代时第一次被加入到 NULL。那么在第 $m+1$ 次迭代, 因为下面这条规则, A 也被加入到 NULL 中了。 ■

$$A \Rightarrow A_1 A_2 \cdots A_i$$

由某个文法产生的语言包含空串, 那么这个空串就能由文法的开始符推导出来, 也就是说, 开始符是可空的。因此, 算法 4.2.1 提出了一种方法, 来判断某个文法的语言中是否包含空串。

例 4.2.1 文法 G 的可空变量集可以由算法 4.2.1 获得。

$$\begin{aligned}
 G: S &\rightarrow ACA \\
 A &\rightarrow aAa \mid B \mid C \\
 B &\rightarrow bB \mid b \\
 C &\rightarrow cC \mid \lambda
 \end{aligned}$$

我们可以通过每次迭代后 NULL 集和 PREV 集的元素来详细叙述算法的执行过程。第 0 次迭代的 NULL 值就是循环之前的值。经过三次迭代, 算法终止了。 G 的可空变量集包括 S 、 A 和 C 。既然开始符是可空的, 那么 $L(G)$ 中包含空串。 □

循环	NULL	PREV
0	{C}	
1	{A, C}	{C}
2	{S, A, C}	{A, C}
3	{S, A, C}	{S, A, C}

109

含有 λ -规则的语法不是非收缩性文法。要建立一个等价的非收缩性文法, 需要加入一些规则。这些规则是用来生成那些原来要靠 λ -规则才能生成的字符串。在规则 $A \Rightarrow uBv$ 的作用下, 可空变量 B 可以有两个作用: 它可以生成一个非空终结符, 或者产生空串。在本例中, 可以这样推导:

$$\begin{aligned}
 A &\Rightarrow uBv \\
 &\Rightarrow uv \\
 &\Rightarrow w.
 \end{aligned}$$

字符串 w 可以不靠 λ -规则生成, 而只要在文法中加入规则 $A \rightarrow w$ 。引理 4.1.2 保证了加入这个规则不会影响文法产生的语言。

规则 $A \rightarrow BABa$ 要增加三个规则, 不然不可需要 λ -规则。如果规则右边的两个 B 都产生空串, 那么需要加入规则 $A \rightarrow Aa$ 。如果要建立一个非收缩性文法, 一共需要 4 条规则才能产生所有由规则 $A \rightarrow BABa$ 产生的串。

$$\begin{aligned}
 A &\rightarrow BABa \\
 A &\rightarrow ABa \\
 A &\rightarrow BAa \\
 A &\rightarrow Aa
 \end{aligned}$$

既然这些规则的右侧都是由 A 推导出的, 那么加入这些规则就不会影响文法的语言。

这种方法可以用来把一个文法 G 转换成不包含 λ -规则的等价文法。如果 $L(G)$ 包含空串, 那么就没有等价的非限定性文法。在推导 $S \Rightarrow \lambda$ 的过程中, 所有出现过的变量最终必然被消灭了。为了处理这种情况, 可以将规则 $S \rightarrow \lambda$ 加入到新文法中, 但是其他所有的 λ -规则都被取代了。如果不考虑 $S \rightarrow \lambda$ 的情况, 这个推导是非收缩的。满足这些条件的文法被称为本质上非收缩性文法 (essentially noncontracting)。

构造等价文法的时候, 在文法下面加下标, 以示区别。文法 G 去除 λ -规则的对等文法表示为 G_1 。

[110]

定理 4.2.3 设 $G = (V, \Sigma, P, S)$ 是一个上下文无关文法。存在构造上下文无关文法 $G_L = (V_L, \Sigma, P_L, S_L)$ 的算法。 G_L 满足下列条件:

- i) $L(G_L) = L(G)$ 。
- ii) S_L 不是一个递归变量。
- iii) 如果 $\lambda \in L(G)$, 那么除了 $S \rightarrow \lambda$, G_L 不包含 λ -规则。否则 G_L 不包含任何 λ -规则。

证明: 开始符可以通过引理 4.1.1 来变成非递归变量。如果引入了新的开始符, 变量集合 V_1 仅仅是 V 加上新的开始符, 否则 V_L 和 V 一样。 G_L 的规则集合 P_L 可以通过两步得到。

1. 对于 P 中的每个规则 $A \rightarrow w$, 如果 w 可以表示为:

$$w = A_1 w_1 A_2 \cdots w_k A_k w_{k+1},$$

其中 A_1, A_2, \dots, A_k 是 w 中可空变量集的子集。那么在 P_L 中加入规则

$$A \rightarrow w_1 w_2 \cdots w_k w_{k+1}.$$

2. 去掉 P_L 中除了 $S \rightarrow \lambda$ 以外的 λ -规则。

第一步根据原始文法中的每条规则中来产生 P_L 的规则。如果一条规则的右侧出现了 n 个可空变量, 则会产生 2^n 条规则。第二步删除了 P_L 中除了 $S \rightarrow \lambda$ 的 λ -规则。 P_L 中的规则要么直接来自 P , 要么从 P 中推导得出。因此 $L(G_L) \subseteq L(G)$ 。

另一方面, 要证明 $L(G)$ 中每一个串都包含在 $L(G_L)$ 中。这是因为所有由 G 中变量推导出的可空串也可以从 G_L 中那个变量推导出。设 $A \xRightarrow{G} w$ 是 G 中的一个推导, $w \in \Sigma^*$ 。可以用归纳法证明 $A \xRightarrow{G_L} w$ 。对推导长度 n 做归纳, 如果 $n=1$, 那么 $A \rightarrow w$ 是 P 中的规则。既然 $w \neq \lambda$, 那么 $A \rightarrow w$ 在 P_L 中。

假设那些在 G 中可以由某个变量通过 n 步或小于 n 步推导出来的串也可以由 G_L 的某个变量中推导出来。在这个假设中没有对该串在 G_L 中的推导长度加以限制。假设 $A \xRightarrow{G_L} w$ 是一个串的推导过程。如果我们使用第一条规则, 则推导过程可以写成:

$$A \Rightarrow w_1 A_1 w_2 A_2 \cdots w_k A_k w_{k+1} \xRightarrow{n} w,$$

这里 $A_i \in V$ 而且 $w_i \in \Sigma^*$ 。根据引理 3.1.5, w 可以表示为:

$$w = w_1 p_1 w_2 p_2 \cdots w_k p_k w_{k+1},$$

[111]

其中 A_i 经过最多 n 步在 G 中推导出 p_i 。对于每个 $p_i \in \Sigma^*$, 归纳假设保证了存在推导 $A_i \xRightarrow{G} p_i$ 。如果 $p_i = \lambda$, 那么 A_i 是 G 的可空变量。在第一步中, 产生了规则:

$$A \rightarrow w_1 A_1 w_2 A_2 \cdots w_k A_k w_{k+1}$$

在这条规则里每个可以推导出空串的变量 A_i 都被去掉了。从 G_L 中推导出 w 可以分两步, 首先使用这条规则, 然后使用归纳假设推导每一个 $p_i \in \Sigma^*$ 。

例 4.2.2 设 G 是例 4.2.1 提供的文法, G 的可空变量集是 $\{S, A, C\}$ 。等价的非收缩性文法 G_L 如下所示:

$$\begin{array}{ll} G: S \rightarrow ACA & G_L: S \rightarrow ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \lambda \\ A \rightarrow aAa \mid B \mid C & A \rightarrow aAa \mid aa \mid B \mid C \\ B \rightarrow bB \mid b & B \rightarrow bB \mid b \\ C \rightarrow cC \mid \lambda & C \rightarrow cC \mid c. \end{array}$$

规则 $S \rightarrow A$ 可以使用两种方法由规则 $S \rightarrow ACA$ 得来的: 去掉前面的 A 和 C , 或者去掉后面的 C 和 A 。所

有的 λ -规则, 除了 $S \rightarrow \lambda$, 都被去掉。 \square

虽然文法 G_L 和 G 等价, 但是在这两个文法中, 同一个串的推导过程可能不一样。最简单的例子是空串。在例 4.2.2 中, G 需要用 6 条规则才能从开始符推导出空串, 而在 G_L 中可以直接就可以推导出空串。这两个文法都可以通过最左推导得到串 aba 。

$$\begin{array}{ll} G: S \Rightarrow ACA & G_L: S \Rightarrow A \\ \Rightarrow aAaCA & \Rightarrow aAa \\ \Rightarrow aBaCA & \Rightarrow aBa \\ \Rightarrow abaCA & \Rightarrow aba \\ \Rightarrow abaA & \\ \Rightarrow abaC & \\ \Rightarrow aba & \end{array}$$

[112] G_L 中第一条推导得到了变量 A , A 最终推导出了所有终结字符串。因此避免了使用 λ -规则。

例 4.2.3 文法 G 产生串 $a^*b^*c^*$:

$$\begin{array}{l} G: S \rightarrow ABC \\ A \rightarrow aA \mid \lambda \\ B \rightarrow bB \mid \lambda \\ C \rightarrow cC \mid \lambda \end{array}$$

G 的可空变量是 S 、 A 、 B 和 C 。去掉 λ -规则的等价文法是

$$\begin{array}{l} G_L: S \rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid b \\ C \rightarrow cC \mid c. \end{array}$$

推导时采用的第一条规则决定了最终会产生哪种终结符。既然 S 是可空的, 那么就加入规则 $S \rightarrow \lambda$ 。 \square

4.3 去掉链规则

在推导中, 使用规则 $A \rightarrow B$ 并不会增加导出的串的长度, 也不会产生新的终结符, 它只是简单地把变量重新命名了。这种规则被称为链规则 (chain rules)。因为链规则仅仅是给变量重命名, 所以可以去掉。考虑以下规则:

$$\begin{array}{l} A \rightarrow aA \mid a \mid B \\ B \rightarrow bB \mid b \mid C. \end{array}$$

链规则 $A \rightarrow B$ 表明所有可以从 B 推导出的串也可以从 A 推导出。对于每条规则 $B \rightarrow w$, 如果可以加入一条规则 $A \rightarrow w$, 那么就可以删除链规则。链规则 $A \rightarrow B$ 可以被三条 A 规则代替, 产生等价的规则

$$\begin{array}{l} A \rightarrow aA \mid a \mid bB \mid b \mid C \\ B \rightarrow bB \mid b \mid C. \end{array}$$

[113] 不幸的是, 生成的新规则中包括了一个新的链规则。可以重复这个过程来去掉新的链规则。除了这样, 我们有一种新的方法可以一次去掉所有的链规则。

如果一个推导 $A \Rightarrow C$ 只包括链规则, 那么称之为链 (chain)。算法 4.3.1 给出了一个非收缩性文法的变量 A 通过链导出的所有变量的生成过程。这个集合表示为 $\text{CHAIN}(A)$ 。集合 NEW 包含那些在前面的迭代中加入到 $\text{CHAIN}(A)$ 中的变量。

算法 4.3.1

集合 $\text{CHAIN}(A)$ 的构造

输入: 本质上的非收缩上下文无关文法 $G = (V, \Sigma, P, S)$

1. $\text{CHAIN}(A) := \{A\}$

2. $\text{PREV} := \emptyset$

3. repeat

3.1. NEW := CHAIN(A) - PREV

3.2. PREV := CHAIN(A)

3.3. for 任意变量 $B \in \text{NEW}$ dofor 任意规则 $B \rightarrow C$ doCHAIN(A) := CHAIN(A) \cup {C}

until CHAIN(A) = PREV

算法 4.3.1 和产生可空变量的算法完全不同。后者首先找到那些直接产生空串的变量，然后再将规则倒推回去。如果规则右边都是 NULL 中的变量，那么左边的变量也会加入到集合中。

产生 CHAIN(A) 的算法遵循自顶向下法则。使用循环递归地建立所有 A 通过链可以推导出的变量。每次循环是在以前发现的链基础上再添上一个规则。算法 4.3.1 中产生 CHAIN(A) 的证明留做练习。

引理 4.3.2 已知 $G = (V, \Sigma, P, S)$ 是一个本质上非收缩上下文无关文法。算法 4.3.1 仅仅使用链规则就由 A 推导出变量集合。

CHAIN(A) 中的变量决定了去掉 A 链规则必须进行的替换。通过去掉 G 中的链规则得到的文法记作 G_c 。

定理 4.3.3 已知 $G = (V, \Sigma, P, S)$ 是本质上非收缩上下文无关文法。那么存在一个算法来构造上下文无关文法 G_c ，使得

i) $L(G_c) = L(G)$ 。ii) G_c 是本质上非收缩的，并且没有链规则。

114|

证明： G_c 中的规则是使用集合 CHAIN(A) 和 G 中的规则构造而成。如果变量 B 和字符串 w 满足下面的条件，则规则 $A \rightarrow w$ 属于 P_c 。

i) $B \in \text{CHAIN}(A)$ ii) $B \rightarrow w \in P$ iii) $w \notin V$

条件 (iii) 保证 P_c 不包含链规则。 G_c 中的变量、字母表以及开始符号都和 G 中的相同。

根据引理 4.1.2， G_c 中可以推导出的每个字符串都可以在 G 中推导出。因此， $L(G_c) \subseteq L(G)$ 。现在已知 $w \in L(G)$ ，并且 $A \xrightarrow{\gamma} B$ 是推导 w 的过程中使用的最长的链规则。w 的推导具有下面的形式

$$S \xRightarrow{\gamma} uAv \xRightarrow{\gamma} uBv \Rightarrow uv \xRightarrow{\gamma} w,$$

其中， $B \rightarrow v$ 是规则，但不是链规则。规则 $A \rightarrow v$ 可以用来替换推导中的链规则序列。重复使用这些技术，去掉所有的链规则的应用，从而最终产生 G_c 中的 w。

例 4.3.1 文法 G_c 是由例 4.2.2 中的 G_1 构造得到的。因为 G_1 是本质上非收缩的，所以算法 4.3.1 使用链规则产生了可以推导出的变量。计算构造集合

$$\text{CHAIN}(S) = \{S, A, C, B\}$$

$$\text{CHAIN}(A) = \{A, B, C\}$$

$$\text{CHAIN}(B) = \{B\}$$

$$\text{CHAIN}(C) = \{C\}.$$

这些集合用来生成 G_c 的规则

$$P_c: S \rightarrow ACA \mid C\bar{A} \mid AA \mid AC \mid aAa \mid aa \mid bB \mid b \mid cC \mid c \mid \lambda$$

$$A \rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cC \mid c$$

□

去掉链规则会增加文法中规则的数目，可是却减少了推导的长度。构造本质上非收缩文法也会碰到这样的折衷。这些限定需要那种能够产生语言，但不会简化推导的附加规则。

115

从本质上非收缩文法中去掉链规则, 它的非收缩性质仍然得到保持。设 $A \rightarrow w$ 是一个由去掉的链规则创建的规则, 这就意味着对于某个变量 $B \in \text{CHAIN}(A)$, 存在规则 $B \rightarrow w$ 。因为原来的文法是本质上非收缩的, 所有唯一的 λ 规则就是 $S \rightarrow \lambda$ 。开始符是非递归的, 对于任意 $A \neq S$, 它都不是 $\text{CHAIN}(A)$ 中的成员。它也遵循构造 P_C 过程中不会增加新的 λ 规则的原则。

没有链规则的本质非收缩文法中的每条规则都具有如下的形式:

- i) $S \rightarrow \lambda$
- ii) $A \rightarrow a$; 或者
- iii) $A \rightarrow w$

其中 $w \in (V \cup \Sigma)^+$, 它的长度至少为 2。规则 $S \rightarrow \lambda$ 仅仅用在空字符串的推导中。使用其他规则就会给待推导的字符串增加终结符, 或者增加字符串的长度。

4.4 无用符

文法用来生成语言, 变量在字符串的产生过程中定义句型的结构。理想情况下, 文法中的每个变量都应该对于生成语言的字符串有贡献。然而, 庞大的文法的构造、对已有文法的修改等都会导致那些生成终结字符串的过程中出现不需要的变量。考察下面的文法:

$$\begin{aligned} G: S &\rightarrow AC \mid BS \mid B \\ A &\rightarrow aA \mid aF \\ B &\rightarrow CF \mid b \\ C &\rightarrow cC \mid D \\ D &\rightarrow aD \mid BD \mid C \\ E &\rightarrow aA \mid BSA \\ F &\rightarrow bB \mid b. \end{aligned}$$

什么是 $L(G)$? 这些变量都出现在终结字符串的产生过程中了吗? 如果是的话, 为什么呢? 试图说服你自己 $L(G) = b^+$ 。为了识别和去掉无用符, 我们首先给出下面的定义。

定义 4.4.1 已知 G 是上下文无关文法。如果存在下面的推导, 那么字符 $x \in (V \cup \Sigma)$ 就是有用的 (useful)。

$$S \xRightarrow{G} uxv \xRightarrow{G} w,$$

116

其中, $u, v \in (V \cup \Sigma)^*$ 并且 $w \in \Sigma^+$ 。没有用处的字符叫做无用的 (useless)。

如果终结符出现在语言 G 的字符串中, 那么这个终结符就是有用的。变量是有用的要满足下面两个条件: 首先, 变量必须出现在文法的句型中 (即: 它必须出现在由 S 推导出的字符串中)。其次, 句型中的每个字符都必须能够推导出终结字符串 (空串也是终结字符串)。下面的两步程序给出了消除无用符的过程。每一步构造满足变量是有用的一个条件。

算法 4.4.2 构造包含可以推导出终结字符串的变量的集合 TERM。算法中使用的策略类似于确定文法中的可空变量集合的策略。算法 4.4.2 的证明过程使用了引理 4.4.2 中的证明策略来生成期望集合, 这个证明过程留作练习。

算法 4.4.2

构造可推导出终结字符串的变量的集合

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

1. TERM := $\{A \mid \text{对于 } w \in \Sigma^+, \text{ 存在一个规则 } A \rightarrow w \in P\}$

2. repeat

2.1. PREV := TERM

2.2. for 任意 $A \in V$ do

if 存在一个 A 规则 $A \rightarrow w$ 且 $w \in (\text{PREV} \cup \Sigma)^+$ then

TERM := TERM $\cup \{A\}$

until PREV = TERM

算法结束时, TERM 包含了 G 中产生终结字符串的变量。不属于 TERM 的变量是无用的。它们对于生成 $L(G)$ 中的字符串是没有贡献的。这个观察结果为我们构造等价于 G 的文法 G_T ——只包含一个推导出终结字符串的变量——提供了动机。

引理 4.4.3 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法。存在一种算法构造上下文无关文 $G_T = (V_T, \Sigma_T, P_T, S)$, 使得

- i) $L(G_T) = L(G)$ 。
- ii) G_T 中的每个变量都可以推导出 G_T 中的每个终结字符串。

证明: P_T 是通过对掉 G 中不能推出终结字符串的变量的所有规则获得的。即: 所有包含变量 V-TERM 的规则。 G_T 的构成成分是

$$V_T = \text{TERM}$$

$$P_T = \{A \rightarrow w \mid A \rightarrow w \text{ 是 } P \text{ 中的规则, } A \in \text{TERM, 并且 } w \in (\text{TERM} \cup \Sigma)^* \text{ 并且}$$

$$\Sigma_T = \{a \in \Sigma \mid a \text{ 出现在 } P_T \text{ 规则的右侧}\}$$

字母表 Σ_T 包含出现在 P_T 规则中的所有终结符。

我们必须证明 $L(G_T) = L(G)$ 。因为 $P_T \subseteq P$, 所以 G_T 中的每个推导也是 G 中的推导, 并且 $L(G_T) \subseteq L(G)$ 。为了构造相对的包含关系, 我们必须证明去掉包含 V-TERM 中的变量的规则, 对于产生终结字符串的集合没有影响。已知 $S \xrightarrow{*} w$ 是字符串 $w \in L(G)$ 的一种推导。这也是 G_T 中的一种推导。如果不是的话, V-TERM 中的变量就必须出现在推导的中间步骤中。由包含在 V-TERM 中的变量构成的句型的推导不能产生终结字符串。因此, 所有推导中使用的规则都属于 P_T , 因此 $w \in L(G_T)$ 。 ■

例 4.4.1 文法 G_T 是由本节开始的文法 G 构造的。

$$\begin{aligned} G: & S \rightarrow AC \mid BS \mid B \\ & A \rightarrow aA \mid aF \\ & B \rightarrow CF \mid b \\ & C \rightarrow cC \mid D \\ & D \rightarrow aD \mid BD \mid C \\ & E \rightarrow aA \mid BSA \\ & F \rightarrow bB \mid b \end{aligned}$$

算法 4.4.2 用来确定 G 中推导出终结字符串的变量。

使用集合 TERM 来构造 G_T 。每轮

$$V_T = \{S, A, B, E, F\}$$

$$\Sigma_T = \{a, b\}$$

$$P_T: S \rightarrow BS \mid B$$

$$A \rightarrow aA \mid aF$$

$$B \rightarrow b$$

$$E \rightarrow aA \mid BSA$$

$$F \rightarrow bB \mid b.$$

迭代	TERM	PREV
0	{B, F}	
1	{B, F, A, S}	{B, F}
2	{B, F, A, S, E}	{B, F, A, S}
3	{B, F, A, S, E}	{B, F, A, S, E}

这个使用变量 C 或 D 得到的间接递归推导, 是使用算法发现的。所有包含这些变量的规则都要删掉。 □

G_T 的构造完成了去掉无用变量过程的第一步。 G_T 中的所有变量都可以推导出终结字符串。我们现在必须去掉那些不会出现在文法的句型中的变量。集合 REACH 就是那些由 S 可以推导出的所有变量构成的。

算法 4.4.4

可达变量集合的构造

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

```

1. REACH := {S}
2. PREV := ∅
3. repeat
    3.1. NEW := REACH - PREV
    3.2. PREV := REACH
    3.3. for 任意  $A \in \text{NEW}$  do
        for 任意规则  $A \rightarrow w$  do 把  $w$  中的所有变量添加到 REACH 中
until REACH = PREV

```

算法 4.4.4 和算法 4.4.3 类似, 使用了自顶向下的方法来构造变量的期望集合。集合 REACH 使用 S 初始化。每当在 S 推导过程中发现了新的变量, 就把它放到 REACH 中。

引理 4.4.5 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法。算法 4.4.4 产生了所有由 S 可以推导出的变量的集合。

证明: 第一, 我们必须证明 REACH 中的每个变量都可以由 S 推导出。证明是依靠对算法迭代的次数进行归纳的。

集合 REACH 初始是 S , 它显然是可达的。假设集合 REACH 中的所有变量经过 n 步迭代都可以由 S 到达。已知 B 是在第 $n+1$ 步迭代添加到集合 REACH 中的。那么存在规则 $A \rightarrow uBV$, 其中, A 经过 n 次迭代就属于 REACH。根据归纳, 存在推导 $S \Rightarrow^* xAv$ 。使用 $A \rightarrow uBV$ 扩展这个推导, 我们就构造了 B 的可达性。

我们现在必须证明由 S 可达的每个变量最终都会添加到集合 REACH 中。如果 $S \Rightarrow^* xAv$, 那么 A 就在第 n 步循环或之前添加到集合 REACH 中。这个证明是通过对由 S 开始的推导的长度进行归纳完成的。

119 开始符是推导长度为 0 的唯一一个可达变量。它在算法的第一步就被放到 REACH 中了。假设推导长度不超过 n 的变量在经过 n 次迭代或之前就放到了 REACH 中。

已知 $S \xRightarrow{n} xAv \Rightarrow^* uBv$, S 是 G 中的推导, 并且第 $n+1$ 步应用的规则是 $A \rightarrow uBV$ 。根据归纳假设, A 在迭代 n 次后放到 REACH 中。在后面的迭代中, B 放到 REACH 中。 ■

定理 4.4.6 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法。存在一个算法来构造上下文无关文法 G_U , 使得

- i) $L(G_U) = L(G)$ 。
- ii) G_U 没有无用符。

证明: 去掉无用符开始于由 G 构造 G_1 。算法 4.4.4 用来生成从开始符可达的 G_1 中的变量。 G_1 中所有引用从 S 不可达的变量的规则都要去掉从而获得 G_1 , 定义为 $V_1 = \text{REACH}$, $P_1 = \{A \rightarrow w \mid A \rightarrow w \in P, A \in \text{REACH}, \text{并且 } w \in (\text{REACH} \cup \Sigma)^*\}$ 而且 $\Sigma_U = \{a \in \Sigma \mid a \text{ 出现在 } P_U \text{ 规则右侧}\}$ 。

为了构造 $L(G_1)$ 和 $L(G_U)$ 的等价性, 需要证明 G_1 可以推导出的每个字符串都可以在 G_U 中推导出。已知 w 是 $L(G_1)$ 中的元素。在推导 w 的过程中出现的每个变量都是可以推导出的, 并且这个过程中使用的规则都属于 P_U 。 ■

例 4.4.2 文法 G_U 是由例 4.4.1 中的 G_1 构造得到的。 G_1 中的可达变量集合可以使用算法 4.4.4 得到。

去掉所有对变量 A 、 E 和 F 的引用而产生的文法是

$$G_U: S \rightarrow BS \mid B \\ B \rightarrow b.$$

循环	REACH	PREV	NEW
0	{S}	∅	
1	{S, B}	{S}	{S}
2	{S, B}	{S, B}	{B}

文法 G_U 等价于本节初给出的文法 G 。很明显, 这些文法产生的语言就是 b^+ 。 □

120 去掉无用符的过程包括定理 4.4.6 中列出的两个步骤。第一步是去掉不能够产生终结字符串的变量。然后清除那些不能够从开始符推导出的变量, 就得到了需要的文法。如果按照相反的顺序应用这

两个步骤,那么可能不会去掉所有的无用符,下一个例子就描述了这种情况。

例 4.4.3 已知 G 是文法

$$G: S \rightarrow a \mid AB$$

$$A \rightarrow b.$$

通过使用两种不同的顺序应用这两个步骤,我们知道了转换过程中按顺序应用步骤的必要性。

去掉不能产生终结字符串的变量:

$$S \rightarrow a$$

$$A \rightarrow b$$

去掉不可达字符:

$$S \rightarrow a$$

去掉不可达字符:

$$S \rightarrow a \mid AB$$

$$A \rightarrow b$$

去掉不能产生终结字符串的变量:

$$S \rightarrow a$$

$$A \rightarrow b$$

变量 A 和终结符 b 是无用的,但是按照相反的顺序进行转换,它们就会仍然留在文法当中。□

文法向范式的转换包含了一系列的算法步骤,每一步的转换仍然保持转换前的性质,去掉无用符的过程不能够取消构造 G_L 或 G_C 获得的限制。这些转换只去掉了规则,而并没有改变文法的任何特点,然而,在把一种文法转化成另一种文法的过程中,可能会引入无用符。这种现象将会在练习 8 和练习 17 的转化中得到解释。

4.5 乔姆斯基范式

范式是使用文法必须满足的条件集合来描述的,乔姆斯基范式对规则右侧的长度和分解进行了限制。 [121]

定义 4.5.1 如果文法的每条规则都符合下面的形式之一,那么上下文无关文法 $G = (V, \Sigma, P, S)$ 就是乔姆斯基范式 (chomsky normal form)。

i) $A \rightarrow BC,$

ii) $A \rightarrow a,$ 或者

iii) $S \rightarrow \lambda,$

其中, $B, C \in V - \{S\}$ 。

因为规则右侧的字符数目最多是 2,所以乔姆斯基范式的文法的推导对应的推导树是二叉树,规则 $A \rightarrow BC$ 的应用就会生成一个子节点是 B 和 C 的节点,所有其他规则的应用都会产生一个子节点。第 7 章将使用二叉推导树表示的推导,来构造上下文无关语言的重复性质。在下一节中,我们将使用这种把文法 G 转化成乔姆斯基范式的能力来获得 $L(G)$ 中字符串的确定成员关系的程序。

在前一节给出的一系列修改之后,我们接着介绍把文法转化成乔姆斯基范式。我们假设要转化的文法 G 有一个非递归的开始符,除了 $S \rightarrow \lambda$ 外没有 λ 规则,没有链规则,并且没有无用符。

定理 4.5.2 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法,存在一种算法来构造等价于 G 的乔姆斯基范式的文法 $G' = (V', \Sigma, P', S')$ 。

证明: 按照前面的转化之后,规则就具有了 $S \rightarrow \lambda$ 、 $A \rightarrow a$ 或 $A \rightarrow w$ 的形式,其中 $w \in ((V \cup \Sigma) - \{S\})^*$, 并且 $\text{length}(w) > 1$ 。 G' 的规则集合 P' 是由 G 的规则构造而成的。

G 中右侧长度为 0 的规则就是 $S \rightarrow \lambda$ 。因为 G 不包含链规则,所以当 w 的长度是 1 时,规则 $A \rightarrow w$ 的右侧是一个终结符,在任何一种情况下,规则都满足乔姆斯基范式的条件,并且可以放到 P' 中。

已知 $A \rightarrow w$ 是 $\text{length}(w)$ 大于 1 的规则。字符串 w 可能既包含变量,也包含终结符。第一步是去掉所有规则右侧的终结符。这一点是通过增加新的变量和规则来完成的,即把每个终结符重新命名为一个变量。例如,规则

可以由下面三条规则替换

$$A \rightarrow bDcF$$

$$A \rightarrow B'DC'F$$

$$B' \rightarrow b$$

$$C' \rightarrow c.$$

右侧的长度等于或超过 2 的规则依然按照这种方式进行转换。这样，规则的右侧就只包含空字符串、终结符和变量串。最后这种情况的规则必须拆成一系列的规则，每个规则的右侧只包含两个变量。应用这一系列的规则，就自然可以生成原来规则的右面。继续前面的例子，我们使用下面的规则替换 A 规则

$$A \rightarrow B'T_1$$

$$T_1 \rightarrow DT_2$$

$$T_2 \rightarrow C'F.$$

变量 T_1 和 T_2 被引入用来连接一系列的规则。重写那些右侧长度大于 2 的规则，从而完成到乔姆斯基范式的转换。

例 4.5.1 已知 G 是文法

$$S \rightarrow aABC \mid a$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bcB \mid bc$$

$$C \rightarrow cC \mid c.$$

这个文法已经满足了开始符、 λ 规则和不包含链规则和空符的条件。通过转换那些右侧长度超过 2 的规则把它转化成等价的乔姆斯基范式文法。

$$G': S \rightarrow A'T_1 \mid a$$

$$A' \rightarrow a$$

$$T_1 \rightarrow AT_2$$

$$T_2 \rightarrow BC$$

$$A \rightarrow A'A \mid a$$

$$B \rightarrow B'T_3 \mid B'C'$$

$$T_3 \rightarrow C'B$$

$$C \rightarrow C'C \mid c$$

$$B' \rightarrow b$$

$$C' \rightarrow c$$

例 4.5.2 规则

$$X \rightarrow aXb \mid ab$$

产生了字符串 $\{a^i b^i \mid i \geq 1\}$ 。增加一个开始符，以及规则 $S \rightarrow X$ ，并且去掉链规则，就产生了新的文法

$$S \rightarrow aXb \mid ab$$

$$X \rightarrow aXb \mid ab.$$

乔姆斯基范式

$$S \rightarrow AT \mid AB$$

$$T \rightarrow XB$$

$$X \rightarrow AT \mid AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

是通过增加规则 $A \rightarrow a$ 和 $B \rightarrow b$ 来获得的。这两条规则给终结符提供了别名，并化简了 S 和 X 规则右侧的长度。

4.6 CYK 算法

已知上下文无关文法 G 和字符串 u , u 属于 $L(G)$ 吗? 这个问题叫做上下文无关文法的成员问题。使用乔姆斯基范式文法中规则的结构, J. Cocke, D. Younger 和 T. Kasami 独立地提出了解决这一问题的算法。CYK 算法使用自底向上的方法来确定字符串的可推导性。

已知 $u = x_1 x_2 \cdots x_n$ 是用来测试成员关系的字符串, 并且 $x_{i,j}$ 表示 u 的子串 $x_i \cdots x_j$ 。值得注意的是, 子串 $x_{i,i}$ 简化 x_i —— u 中第 i 个字符。CYK 算法的策略是:

- 第一步: 对于 u 中长度为 1 的每个子串 $x_{i,i}$, 找到规则 $A \rightarrow x_{i,i}$ 中的 A 这种变量构成的集合 $X_{i,i}$ 。
- 第二步: 对于 u 中长度为 2 的每个子串 $x_{i,i+1}$, 找到产生初始推导 $A \Rightarrow x_{i,i+1}$ 中的变量构成的集合 $X_{i,i+1}$ 。
- 第三步: 对于 u 中长度为 3 的每个子串 $x_{i,i+2}$, 找到产生初始推导 $A \Rightarrow x_{i,i+2}$ 中的变量构成的集合 $X_{i,i+2}$ 。
- 第 $n-1$ 步: 对于 u 中长度为 $n-1$ 的子串 $x_{1,n-1}$, $x_{2,n}$, 找到产生初始推导 $A \Rightarrow x_{1,n-1}$ 和 $A \Rightarrow x_{2,n}$ 中变量构成的集合 $X_{1,n-1}$ 和 $X_{2,n}$ 。
- 第 n 步: 对于 u 中所有长度为 n 的字符串 $x_{1,n} = u$, 找到产生初始推导 $A \Rightarrow x_{1,n}$ 中所有变量构成的集合 $X_{1,n}$ 。

如果开始符 S 属于 $X_{1,n}$, 那么 u 就属于这种文法产生的语言。集合 $X_{i,j}$ 的生成过程使用了一个叫做动态编程的技术。动态编程的重要特点就是在第 t 步计算集合 $X_{i,j}$ 需要的信息都已经在第 1 到 $t-1$ 步获得了。

我们看看为什么这种性质对于使用乔姆斯基范式的推导是正确的。在第一步构造集合的过程是很直观的。如果 $A \rightarrow x_i$ 是文法的规则, 那么 $A \in X_{i,i}$ 。

对于第二步, 子串 $x_{i,i+1}$ 的推导具有如下形式

$$\begin{aligned} A &\Rightarrow BC \\ &\Rightarrow x_i C \\ &\Rightarrow x_i x_{i+1} \end{aligned}$$

因为 B 推导出 x_i , 而 C 推导出 x_{i+1} , 所以这些变量都应该属于 $X_{i,i}$ 和 $X_{i+1,i+1}$ 。当存在规则 $A \rightarrow BC$, 并且 $B \in X_{i,i}$ 时, 就把变量 A 放到 $X_{i,i+1}$ 中。

现在我们考虑算法第 t 步中的集合 $X_{i,i+t}$ 的产生。我们希望找到推导出子串 $x_{i,i+t}$ 的所有变量。应用第一条规则到这样的推导中, 就产生了两个变量, 分别称之为 B 和 C 。紧接着是用 B 和 C 开始的推导, 产生 $x_{i,i+t}$ 。因此, 推导过程如下

$$\begin{aligned} A &\Rightarrow BC \\ &\Rightarrow x_{i,k} C \\ &\Rightarrow x_{i,k} x_{k+1,i+t} \end{aligned}$$

其中对于任意的介于 i 和 $t-1$ 之间的 k , 都有 B 产生 $x_{i,k}$, C 产生 $x_{k+1,i+t}$ 。因此, A 推导出 $x_{i,i+t}$ 仅当存在规则 $A \rightarrow BC$, 并且数字 k 介于 i 和 $t-1$ 之间, 且 $B \in X_{i,k}$ 和 $C \in X_{k+1,i+t}$ 。在检查这个条件中, 所有需要考察的集合都是在第 t 步之前产生。

集合 $X_{i,j}$ 可以表示成一个 $n \times n$ 矩阵的上三角部分。

	1	2	3	...	$n-1$	n
1	$X_{1,1}$	$X_{1,2}$	$X_{1,3}$...	$X_{1,n-1}$	$X_{1,n}$
2		$X_{2,2}$	$X_{2,3}$...	$X_{2,n-1}$	$X_{2,n}$
3			$X_{3,3}$...	$X_{3,n-1}$	$X_{3,n}$
\vdots				\ddots		\vdots
$n-1$					$X_{n-1,n-1}$	$X_{n-1,n}$
n						$X_{n,n}$

CYK 算法使用对角化从 $X_{1,n}$ 的右上角开始构造了对角的入口。

我们使用例 4.5.2 中生成 $\{a^i b^i \mid i \geq 1\}$ 和字符串 $aaabbb$ 的文法来解释 CYK 算法。表 4-1 跟踪了算法的步骤, 并给出了计算的结果。

	1	2	3	4	5	6
1	{A}	\emptyset	\emptyset	\emptyset	\emptyset	{S, X}
2		{A}	\emptyset	\emptyset	{S, X}	{T}
3			{A}	{S, X}	{T}	\emptyset
4				{B}	\emptyset	\emptyset
5					{B}	\emptyset
6						{B}

对角化的集合是使用规则 $A \rightarrow a$ 和 $B \rightarrow b$ 获得的。第二步产生了对角上的入口。集合 $X_{1,1}$ 的构造仅仅需要考虑子串 $x_{1,1}$ 和 $x_{1,1+1-1}$ 。例如, 如果存在 $X_{1,1} = \{A\}$ 和 $X_{1,2} = \{A\}$ 中的变量, 它们是规则的右侧, 那么变量就属于 $X_{1,2}$ 。因为 AA 不是规则的右侧, 所以 $X_{1,2} = \emptyset$ 。集合 $X_{1,4}$ 是由 $X_{1,1} = \{A\}$ 和 $X_{4,4} = \{B\}$ 构造而成的。字符串 AB 是 $S \rightarrow AB$ 和 $X \rightarrow AB$ 的右侧。因此, S 和 X 都属于 $X_{1,4}$ 。

在第 t 步, 存在待检查的子串 $x_{i,i+1}$ 的 $t-1$ 个不同的分解。表 4-1 中的最右列中的集合 $X_{1,6}$ 是通过检查所有 $t-1$ 种可能性获得的变量的并。例如, 计算 $X_{1,6}$ 需要考察两种分解 $x_{1,4}x_{4,6}$ 和 $x_{1,5}x_{5,6}$ 。因为 $S \in X_{1,4}$, $B \in X_{5,5}$, 所以变量 T 被放到这个集合中, 并且 $T \rightarrow SB$ 是正确的。集合 $X_{1,6}$ 中 S 的存在意味着字符串 $aaabbb$ 属于这种文法产生的语言。

利用前面列出的步骤, 算法 4.6.1 给出了 CYK 对于成员问题的解决方案。围绕着对角化的集合计算参见第二行。变量 $step$ 指的是待分析的子串的长度。在第 3.1 步开始的循环中, i 指的是子串的开始位置, 而 k 指的是第一个和第二个构成成分被拆分的位置。

算法 4.6.1

CYK 算法

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

字符串 $u = x_1 x_2 \cdots x_n \in \Sigma^*$

1. 把所有的 $X_{i,j}$ 初始化为 \emptyset

2. for $i = 1$ to n

对每个变量 A , 如果存在规则 $A \rightarrow x_i$, 那么 $X_{i,i} := X_{i,i} \cup \{A\}$

3. for $step = 2$ to n

3.1. for $i = 1$ to $n - step + 1$

3.1.1. for $k = i$ to $i + step - 2$

如果存在变量 $B \in X_{i,k}$, $C \in X_{k+1,i+step-1}$, 以及

规则 $A \rightarrow BC$, 那么 $X_{i,i+step-1} := X_{i,i+step-1} \cup \{A\}$

4. 如果 $S \in X_{1,n}$, $u \in L(G)$

上面列出的 CYK 算法, 是用来确定一个字符串 u 是否可以由一种乔姆斯基范式文法推导得出。可以对算法进行修改, 从而用来生成 $L(G)$ 中字符串的推导, 即: 分析器。这可以通过记录往集合 $X_{i,j}$ 中添加的变量的调整来完成。为了演示这种方法, 我们将使用表 4-1 中的计算的轨迹来生成字符串 $aaabbb$ 的推导。标有“集合”的列表示包含匹配规则右侧的变量的集合。例如, 变量 S 被放到 $X_{6,6}$ 中, 因为 $A \in X_{1,1}$ 和 $T \in X_{2,6}$ 的存在匹配了规则 $S \rightarrow AT$ 的右侧。把

推导	集合
$S \Rightarrow AT$	$A \in X_{1,1}, T \in X_{2,6}$
$\Rightarrow aT$	$T \in X_{2,6}$
$\Rightarrow aXB$	$X \in X_{2,5}, B \in X_{6,6}$
$\Rightarrow aATB$	$A \in X_{2,2}, T \in X_{3,5}, B \in X_{6,6}$
$\Rightarrow aaTB$	$T \in X_{3,5}, B \in X_{6,6}$
$\Rightarrow aaXBB$	$X \in X_{3,4}, B \in X_{5,5}, B \in X_{6,6}$
$\Rightarrow aaABBB$	$A \in X_{3,3}, B \in X_{4,4}, B \in X_{5,5}, B \in X_{6,6}$
$\Rightarrow aaabbb$	

这个构造过程逆过来, 就可以使用规则 $S \rightarrow A$ 来推导 $aaabbb$ 。

CYK 算法作为分析器的应用受限于需要找到推导的计算需求。对于长度为 n 的输入字符串, 需要构造 $(n^2 + n)/2$ 个集合来完成动态编程表格。而且, 每个这样的集合都需要考虑相关子串的多种分解。在本书的第五部分, 我们考察那些专门针对有效分析而设计的文法和算法。

127

表 4-1 CYK 算法的轨迹

步骤	字符串 $x_{i,j}$	子串	$X_{i,k}$	$X_{k+1,j}$	$X_{i,j}$
2	$x_{1,2} = aa$	$x_{1,1}, x_{2,2}$	$\{A\}$	$\{A\}$	\emptyset
	$x_{2,3} = aa$	$x_{2,2}, x_{3,3}$	$\{A\}$	$\{A\}$	\emptyset
	$x_{3,4} = ab$	$x_{3,3}, x_{4,4}$	$\{A\}$	$\{B\}$	$\{S, X\}$
	$x_{4,5} = bb$	$x_{4,4}, x_{5,5}$	$\{B\}$	$\{B\}$	\emptyset
	$x_{5,6} = bb$	$x_{5,5}, x_{6,6}$	$\{B\}$	$\{B\}$	\emptyset
3	$x_{1,3} = aaa$	$x_{1,1}, x_{2,3}$	$\{A\}$	\emptyset	\emptyset
		$x_{1,2}, x_{3,3}$	\emptyset	$\{A\}$	\emptyset
	$x_{2,4} = aab$	$x_{2,2}, x_{3,4}$	$\{A\}$	$\{S, X\}$	\emptyset
		$x_{2,3}, x_{4,4}$	\emptyset	$\{B\}$	\emptyset
	$x_{3,5} = abb$	$x_{3,3}, x_{4,5}$	$\{A\}$	\emptyset	\emptyset
		$x_{3,4}, x_{5,5}$	$\{S, X\}$	$\{B\}$	$\{T\}$
	$x_{4,6} = bbb$	$x_{4,4}, x_{5,6}$	$\{B\}$	\emptyset	\emptyset
		$x_{4,5}, x_{6,6}$	\emptyset	$\{B\}$	\emptyset
4	$x_{1,4} = aaab$	$x_{1,1}, x_{2,4}$	$\{A\}$	\emptyset	\emptyset
		$x_{1,2}, x_{3,4}$	\emptyset	$\{S, X\}$	\emptyset
		$x_{1,3}, x_{4,4}$	\emptyset	$\{B\}$	\emptyset
	$x_{2,5} = aabb$	$x_{2,2}, x_{3,5}$	$\{A\}$	$\{T\}$	$\{S, X\}$
		$x_{2,3}, x_{4,5}$	\emptyset	\emptyset	\emptyset
		$x_{2,4}, x_{5,5}$	\emptyset	$\{B\}$	\emptyset
		$x_{3,3}, x_{4,6}$	$\{A\}$	\emptyset	\emptyset
	$x_{3,6} = abbb$	$x_{3,4}, x_{5,6}$	$\{S, X\}$	\emptyset	\emptyset
		$x_{3,5}, x_{6,6}$	$\{T\}$	$\{B\}$	\emptyset
5	$x_{1,5} = aaabb$	$x_{1,1}, x_{2,5}$	$\{A\}$	$\{S, X\}$	\emptyset
		$x_{1,2}, x_{3,5}$	\emptyset	$\{T\}$	\emptyset
		$x_{1,3}, x_{4,5}$	\emptyset	\emptyset	\emptyset
		$x_{1,4}, x_{5,5}$	\emptyset	$\{B\}$	\emptyset
	$x_{2,6} = aabbb$	$x_{2,2}, x_{3,6}$	$\{A\}$	\emptyset	\emptyset
		$x_{2,3}, x_{4,6}$	\emptyset	\emptyset	\emptyset
		$x_{2,4}, x_{5,6}$	\emptyset	\emptyset	\emptyset
		$x_{2,5}, x_{6,6}$	$\{S, X\}$	$\{B\}$	$\{T\}$
6	$x_{1,6} = aaabbb$	$x_{1,1}, x_{2,6}$	$\{A\}$	$\{T\}$	$\{S, X\}$
		$x_{1,2}, x_{3,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,3}, x_{4,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,4}, x_{5,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,5}, x_{6,6}$	\emptyset	$\{B\}$	\emptyset

128

4.7 去掉直接左递归

在任意的上下文无关文法的推导中, 可以在推导的任意位置使用规则, 从而按照任意顺序产生终结符。例如, 文法 G_1 中的推导在变量的右侧产生了终结符, 而 G_2 中的推导在两侧都产生了终结符。

$$\begin{array}{ll}
 G_1: S \rightarrow Aa & G_2: S \rightarrow aAb \\
 A \rightarrow Aa \mid b & A \rightarrow aAb \mid \lambda
 \end{array}$$

乔姆斯基范式在推导当中为终结符的产生添加了结构。字符串是按照从左到右的顺序应用每一条规则添加一个终结符构造的。在推导 $S \Rightarrow uAv$ 中, A 是最左变量, 字符串 u 称作是句型的终结符前缀 (terminal prefix)。我们的目的就是构造一种应用每条规则都可以增加终结符前缀的文法。

文法 G_1 提供与我们期望完全相对的规则的例子。变量 A 一直都是最左符号, 直到推导使用规则 $A \rightarrow b$ 结束。考虑字符串 $baaa$ 的推导

$$\begin{aligned} S &\Rightarrow Aa \\ &\Rightarrow Aaa \\ &\Rightarrow Aaaa \\ &\Rightarrow baaa. \end{aligned}$$

应用左递归规则 $A \rightarrow Aa$ 产生了 a 的字符串, 但是并没有增加终结符前缀的长度。这种形式的推导叫做直接左递归 (directly left-recursive)。只有在没有使用左递归规则时, 前缀才增加。

向乔姆斯基范式转化过程中的重要一步就是从文法中去掉左递归规则。下面几个例子解释了消除左递归规则的技术

$$\begin{array}{lll} \text{a) } A \rightarrow Aa \mid b & \text{b) } A \rightarrow Aa \mid Ab \mid b \mid c & \text{c) } A \rightarrow AB \mid BA \mid a \\ & & B \rightarrow b \mid c \end{array}$$

使用这些集合生成的是 ba^* 、 $(b \cup c)(a \cup b)^*$ 和 $(b \cup c)^*a(b \cup c)^*$ 。左递归在递归变量的右侧构造字符串。递归序列是通过应用不是左递归的 A 规则来终止的。为了从左到右地构造字符串, 首先应用非递归规则, 然后使用右递归来构造剩下的字符串。下面的规则产生了和前面的例子相同的字符串, 而且它们没有使用直接左递归。

$$\begin{array}{lll} \text{a) } A \rightarrow bZ \mid b & \text{b) } A \rightarrow bZ \mid cZ \mid b \mid c & \text{c) } A \rightarrow BAZ \mid aZ \mid BA \mid a \\ Z \rightarrow aZ \mid a & Z \rightarrow aZ \mid bZ \mid a \mid b & Z \rightarrow BZ \mid B \\ & & B \rightarrow b \mid c \end{array}$$

(a) 中的规则使用右递归代替左递归, 从而产生了 ba^* 。使用这些规则, 在推导 $baaa$ 的过程中, 应用它们中的每一条规则来增加终结符前缀的长度。

$$\begin{aligned} A &\Rightarrow bZ \\ &\Rightarrow baZ \\ &\Rightarrow baaZ \\ &\Rightarrow baaa \end{aligned}$$

去掉直接左递归, 需要在文法中添加一个新的变量。这个变量引入了一个右递归规则的集合。直接右递归导致递归变量出现在推导字符串的最右端。

为了消除直接左递归, A 规则被拆成两部分: 左递归规则

$$A \rightarrow Au_1 \mid Au_2 \mid \dots \mid Au_j$$

和规则

$$A \rightarrow v_1 \mid v_2 \mid \dots \mid v_k,$$

其中, 每个 v_i 的第一个字符都不是 A 。使用这些规则的最左推导过程包括一系列左递归规则, 然后直接应用规则 $A \rightarrow v_i$ 来结束直接左递归。使用前面例子解释的技术, 我们构造新的规则来先产生 v_i , 然后使用右递归生成字符串的剩下部分。

A 规则

$$A \rightarrow v_1 \mid \dots \mid v_k \mid v_1Z \mid \dots \mid v_kZ$$

首先产生推导出字符串的左侧的一个 v_i 。如果字符串包含一系列的 u_i , 那么它们就可以使用右递归由 Z 规则推导出

$$Z \rightarrow u_1Z \mid \dots \mid u_jZ \mid u_1 \mid \dots \mid u_j$$

例 4.7.1 使用下面这个规则集合, 但是不要使用直接左递归, 来生成同样的字符串。

$$A \rightarrow Aa \mid Aab \mid bb \mid b$$

这些规则产生的是 $(b \cup bb)(a \cup ab)^*$ 在推导中的直接左递归通过应用原来的规则 $A \rightarrow b$ 或 $A \rightarrow bb$ 来终止推导过程。为了按照从左到右的方式来构造这些字符串, 我们使用 A 规则

$$A \rightarrow bb \mid b \mid bbZ \mid bZ$$

来产生字符串的最左字符。 Z 规则使用下面的右递归规则来生成 $(a \cup ab)^*$:

$$Z \rightarrow aZ \mid abZ \mid a \mid ab.$$

□ [130]

引理 4.7.1 已知 $G = (V, \Sigma, P, S)$ 是上下文无关文法, 并且 $A \in V$ 是 G 中的直接左递归变量。那么就存在一种算法来构造等价的文法 $G' = (V', \Sigma, P', S')$, 使得 A 不是直接左递归的。

证明: 我们假设 G 的开始符是非递归的, 惟一的 λ 规则是 $S \rightarrow \lambda$, 并且 P 不包含规则 $A \rightarrow A$ 。如果不是这样的话, G 就可以转化成满足这些条件的等价文法。 G' 的变量是 G 中的变量增加一个附加的变量来生成右递归规则。 P' 是由 P 使用上面列出的技术来构造的。

新的 A 规则不是左递归的, 因为每个 v_i 的第一个字符不是 A 。 Z 规则也不是左递归的。变量 Z 不出现在 u_i 中, 并且根据 G 中的 A 规则 u_i 是非空的。■

这个技术可以反复使用, 从而去掉所有使用左递归规则的地方, 并且获得的还是文法的语言。然而, 使用规则 $A \rightarrow Bu$ 和 $B \rightarrow Av$ 的推导, 可以生成句型

$$\begin{aligned} A &\Rightarrow Bu \\ &\Rightarrow Avu \\ &\Rightarrow Buvu \\ &\Rightarrow Avuvu \\ &\vdots \end{aligned}$$

这就展示了使用直接左递归推导出的终结符前缀的长度的共同缺陷。格立巴赫范式的转换将去掉所有可能的非直接左递归。

4.8 格立巴赫范式

在格立巴赫范式中, 每个规则的应用, 就会给期望字符串的终结符前缀增加一个字符。这就保证了左递归, 不管是直接还是间接的, 都不会出现。它也能保证长度 $n > 0$ 的字符串的推导仅包含 n 步规则应用。

定义 4.8.1 如果每条规则都具有下述形式中的一种, 那么这个上下文无关文法 $G = (V, \Sigma, P, S)$ 就属于格立巴赫范式 (Greibach normal form) 的。

- i) $A \rightarrow aA_1A_2 \cdots A_n$,
- ii) $A \rightarrow a$, 或者
- iii) $S \rightarrow \lambda$,

其中, $a \in \Sigma$ 和对于 $i = 1, 2, \dots, n$ 都有 $A_i \in V - \{S\}$ 。

[131]

乔姆斯基范式转化成格立巴赫范式要使用两个规则转化技术: 引理 4.1.3 中的规则替换方法以及去掉左递归的转化。这个过程首先是给文法中的变量编号、开始符编号 1, 剩下的变量依照任意顺序编号。不同的编号顺序可能会导致不同的文法转换, 但是按照什么顺序转化都可以。

转化的第一步是构造文法, 从而使其中的任何一条规则都具有下面的形式之一:

- i) $S \rightarrow \lambda$
- ii) $A \rightarrow aw$, 或者
- iii) $A \rightarrow Bw$,

其中 $w \in V^*$, 并且在给变量编号的过程中赋予 B 的序号的数字都要比 A 的大。根据变量标号的顺序, 这些规则被转化成满足条件 (iii)。乔姆斯基范式文法转化成格立巴赫范式是通过跟踪文法 G 的规则转化来解释的:

$$\begin{aligned} G: S &\rightarrow AB \mid \lambda \\ A &\rightarrow AB \mid CB \mid a \\ B &\rightarrow AB \mid b \\ C &\rightarrow AC \mid c. \end{aligned}$$

变量 S , A , B 和 C 分别编号为 1, 2, 3 和 4。

因为乔姆斯基范式文法的开始符不是递归的, 所以 S 规则已经满足了这三个条件。继续把 A 规则转化为一组规则的集合, 这些集合右侧的第一个字符或者是终结符, 或者是一个序号大于 2 的变量。左递归规则 $A \rightarrow AB$ 影响了这些限制。引理 4.7.1 用来去掉直接左递归, 从而得到

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow AB \mid b \\ C &\rightarrow AC \mid c \\ R_1 &\rightarrow BR_1 \mid B. \end{aligned}$$

现在 B 规则必须转化成正确的形式。规则 $B \rightarrow AB$ 必须被替换, 因为 B 的序号是 3, 而右侧的第一个字符 A 的序号为 2。引理 4.1.3 允许规则 $B \rightarrow AB$ 右侧的首位 A 被 A 规则的右部替换, 从而产生

132

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\ C &\rightarrow AC \mid c \\ R_1 &\rightarrow BR_1 \mid B. \end{aligned}$$

把引理 4.1.3 的替换技术应用到 C 规则, 从而产生两个左递归规则

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\ C &\rightarrow CBR_1C \mid aR_1C \mid CBC \mid aC \mid c \\ R_1 &\rightarrow BR_1 \mid B \end{aligned}$$

引入新变量 R_2 可以去掉左递归。

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\ C &\rightarrow aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\ R_1 &\rightarrow BR_1 \mid B \\ R_2 &\rightarrow BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC \end{aligned}$$

原来的变量现在满足以下条件: 规则右侧的第一个字符或者是终结符, 或者是序号大于左侧的变量的序号的变量。序号最大的变量, 在这里是 C , 必须在每条规则中有一个终结符作为第一个字符。下一个变量 B , 必须有 C 或者终结符作为它的第一个字符。以变量 C 开始的 B 规则可以被一组规则替换, 根据 C 规则和引理 4.1.3, 这些规则要以终结符作为首字符。完成上述转换, 我们得到

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow aR_1B \mid aB \mid b \\ &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\ &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\ C &\rightarrow aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\ R_1 &\rightarrow BR_1 \mid B \\ R_2 &\rightarrow BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC. \end{aligned}$$

133

B 规则的第二列是通过把规则 $B \rightarrow CBR_1B$ 中的 C 替换得到的, 第三条是替换规则。 S 和 A 规则必须重写从而去掉规则右部初始位置的变量。 A 规则中的替换使用的是 B 规则和 C 规则, 这几条规则都是以

终结符开始的 A 规则、 B 规则和 C 规则都可以用来对 S 规则进行转换,从而生成

$$\begin{aligned} S &\rightarrow \lambda \\ &\rightarrow aR_1B \mid aB \\ &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\ &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \end{aligned}$$

$$\begin{aligned} A &\rightarrow aR_1 \mid a \\ &\rightarrow aR_1CBR_1 \mid aCBR_1 \mid cBR_1 \mid aR_1CR_2BR_1 \mid aCR_2BR_1 \mid cR_2BR_1 \\ &\rightarrow aR_1CB \mid aCB \mid cB \mid aR_1CR_2B \mid aCR_2B \mid cR_2B \end{aligned}$$

$$\begin{aligned} B &\rightarrow aR_1B \mid aB \mid b \\ &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\ &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \end{aligned}$$

$$C \rightarrow aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2$$

$$R_1 \rightarrow BR_1 \mid B$$

$$R_2 \rightarrow BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC.$$

最后, 替换过程必须应用到去掉直接递归中的每个变量上。重写这些规则得到,

$$\begin{aligned} R_1 &\rightarrow aR_1BR_1 \mid aBR_1 \mid bR_1 \\ &\rightarrow aR_1CBR_1BR_1 \mid aCBR_1BR_1 \mid cBR_1BR_1 \mid aR_1CR_2BR_1BR_1 \mid aCR_2BR_1BR_1 \mid cR_2BR_1BR_1 \\ &\rightarrow aR_1CBBR_1 \mid aCBBR_1 \mid cBBR_1 \mid aR_1CR_2BBR_1 \mid aCR_2BBR_1 \mid cR_2BBR_1 \end{aligned}$$

$$\begin{aligned} R_1 &\rightarrow aR_1B \mid aB \mid b \\ &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\ &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \end{aligned}$$

$$\begin{aligned} R_2 &\rightarrow aR_1BR_1CR_2 \mid aBR_1CR_2 \mid bR_1CR_2 \\ &\rightarrow aR_1CBR_1BR_1CR_2 \mid aCBR_1BR_1CR_2 \mid cBR_1BR_1CR_2 \mid aR_1CR_2BR_1BR_1CR_2 \mid aCR_2BR_1BR_1CR_2 \mid \\ &\quad cR_2BR_1BR_1CR_2 \\ &\rightarrow aR_1CBBR_1CR_2 \mid aCBBR_1CR_2 \mid cBBR_1CR_2 \mid aR_1CR_2BBR_1CR_2 \mid aCR_2BBR_1CR_2 \mid cR_2BBR_1CR_2 \end{aligned}$$

134

$$\begin{aligned} R_2 &\rightarrow aR_1BCR_2 \mid aBCR_2 \mid bCR_2 \\ &\rightarrow aR_1CBR_1BCR_2 \mid aCBR_1BCR_2 \mid cBR_1BCR_2 \mid aR_1CR_2BR_1BCR_2 \mid aCR_2BR_1BCR_2 \mid cR_2BR_1BCR_2 \\ &\rightarrow aR_1CBBR_2 \mid aCBBR_2 \mid cBBR_2 \mid aR_1CR_2BBR_2 \mid aCR_2BBR_2 \mid cR_2BBR_2 \end{aligned}$$

$$\begin{aligned} R_2 &\rightarrow aR_1BR_1C \mid aBR_1C \mid bR_1C \\ &\rightarrow aR_1CBR_1BR_1C \mid aCBR_1BR_1C \mid cBR_1BR_1C \mid aR_1CR_2BR_1BR_1C \mid aCR_2BR_1BR_1C \mid cR_2BR_1BR_1C \\ &\rightarrow aR_1CBBR_1C \mid aCBBR_1C \mid cBBR_1C \mid aR_1CR_2BBR_1C \mid aCR_2BBR_1C \mid cR_2BBR_1C \end{aligned}$$

$$\begin{aligned} R_2 &\rightarrow aR_1BC \mid aBC \mid bC \\ &\rightarrow aR_1CBR_1BC \mid aCBR_1BC \mid cBR_1BC \mid aR_1CR_2BR_1BC \mid aCR_2BR_1BC \mid cR_2BR_1BC \\ &\rightarrow aR_1CBBBC \mid aCBBBC \mid cBBBC \mid aR_1CR_2BBBC \mid aCR_2BBBC \mid cR_2BBBC. \end{aligned}$$

格立巴赫范式中的文法失去了原有文法 G 的所有简单性。设计一种格立巴赫范式形式的文法几乎是一项不可能完成的任务。文法的构造应该使用简单直观的规则。从前面的转换过程可以看出, 把任意上下文无关文法转化成格立巴赫范式的过程都可以写成算法, 因此可以使用设计好的计算机程序自动完成。这个程序的输入包括任意上下文无关文法的规则, 其结果是等价的格立巴赫范式文法。

应该指出的是, 使用引理 4.1.3 中的替换规则可能会产生无用符号。变量 A 是 G 的有用符号, 它出现在下面的推导中

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab.$$

在向格立巴赫范式转化的过程中, 替换会去掉规则右侧所有出现的 A 。字符串 ab 由等价的格立巴赫范式按下面的过程生成。

$$S \Rightarrow aB \Rightarrow ab$$

[135]

定理 4.8.2 已知 G 是上下文无关文法。存在一种算法可以用来构造等价的格立巴赫范式形式的上下文无关文法。

证明：用来构造格立巴赫范式的操作已经被证明可以用来产生等价的文法。所有这些就是为了证明这些规则总可以进行转化，从而满足替换必须的条件。这就需要每个规则都有下面的形式

$$A_k \rightarrow A_j w, k < j$$

或

$$A_k \rightarrow aw,$$

其中，下角标表示变量的序号。

对变量的序号进行归纳证明。基础是开始符，变量的序号是 1。因为 S 不是递归的，所以显然成立。现在假设所有序号不超过 k 的变量都满足条件。如果存在规则 $A_i \rightarrow A_j w$ ，并且 $i < k$ ，那么就可以替换 A_i ，从而生成一组规则，并且每条规则都形如 $A_i \rightarrow A_j w'$ ，其中 $j > i$ 。如果必须的话重复这个过程 $k-i$ 次，从而生成一组或者有左递归或者具有正确形式的规则。所有直接左递归的变量都可以使用引理 4.7.1 来转换。 ■

例 4.8.1 乔姆斯基范式和格立巴赫范式使用下面的文法进行构造：

$$S \rightarrow SaB \mid aB$$

$$B \rightarrow bB \mid \lambda.$$

增加一个非递归的开始符 S' ，去掉 λ 和链规则，得到

$$S' \rightarrow SaB \mid Sa \mid aB \mid a$$

$$S \rightarrow SaB \mid Sa \mid aB \mid a$$

$$B \rightarrow bB \mid b.$$

乔姆斯基范式可以通过前面规则的转换获得。变量 A 和 C 分别是 a 和 c 的别名， T 表示字符串 aB 。

$$S' \rightarrow ST \mid SA \mid AB \mid a$$

$$S \rightarrow ST \mid SA \mid AB \mid a$$

$$B \rightarrow CB \mid b$$

$$T \rightarrow AB$$

$$A \rightarrow a$$

$$C \rightarrow b$$

[136]

变量使用 S' ， S ， B ， T ， A 和 C 来编号。去掉左递归 S 规则，得到

$$S' \rightarrow ST \mid SA \mid AB \mid a$$

$$S \rightarrow ABZ \mid aZ \mid AB \mid a$$

$$B \rightarrow CB \mid b$$

$$T \rightarrow AB$$

$$A \rightarrow a$$

$$C \rightarrow b$$

$$Z \rightarrow TZ \mid AZ \mid T \mid A.$$

这些规则满足条件：规则左侧的变量的值要小于右侧第一位置的变量的值。实现由 A 规则和 C 规则开始的替换，则会产生如下的格立巴赫范式：

$$S' \rightarrow aBZT \mid aZT \mid aBT \mid aT \mid aBZA \mid aZA \mid aBA \mid aA \mid aB \mid a$$

$$S \rightarrow aBZ \mid aZ \mid aB \mid a$$

$$B \rightarrow bB \mid b$$

$$T \rightarrow aB$$

$$A \rightarrow a$$

$$C \rightarrow b$$

$$Z \rightarrow aBZ \mid aZ \mid aB \mid a.$$

下面的表格给出三种等价的文法中, 字符串 *abaaba* 的最左推导。

G	乔姆斯基范式	格立巴赫范式
$S \Rightarrow SaB$	$S' \Rightarrow SA$	$S' \Rightarrow aBZA$
$\Rightarrow SaBaB$	$\Rightarrow STA$	$\Rightarrow abZA$
$\Rightarrow SaBaBaB$	$\Rightarrow SATA$	$\Rightarrow abaZA$
$\Rightarrow aBaBaBaB$	$\Rightarrow ABATA$	$\Rightarrow abaaBA$
$\Rightarrow abBaBaBaB$	$\Rightarrow aBATA$	$\Rightarrow abaabA$
$\Rightarrow abaBaBaB$	$\Rightarrow abATA$	$\Rightarrow abaaba$
$\Rightarrow abaaBaB$	$\Rightarrow abaTA$	
$\Rightarrow abaabBaB$	$\Rightarrow abaABA$	
$\Rightarrow abaabaB$	$\Rightarrow abaaBA$	
$\Rightarrow abaaba$	$\Rightarrow abaabA$	
	$\Rightarrow abaaba$	

[137]

乔姆斯基范式文法的推导过程中产生了六个变量。每个变量应用形如 $A \rightarrow a$ 的规则都可以转化成终结符。格立巴赫范式推导使用每个规则产生一个终结符, 它仅仅使用了六个推导过程就完成。□

4.9 练习

从练习 1 到练习 5, 使用非递归的开始符构造等价但不缩小的文法 G_1 。给出每个文法产生的语言的正则表达式。

1. $G: S \rightarrow aS \mid bS \mid B$

$B \rightarrow bb \mid C \mid \lambda$

$C \rightarrow cC \mid \lambda$

2. $G: S \rightarrow ABC \mid \lambda$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid A$

$C \rightarrow cC \mid \lambda$

3. $G: S \rightarrow BSA \mid A$

$A \rightarrow aA \mid \lambda$

$B \rightarrow Bb \mid \lambda$

4. $G: S \rightarrow AB \mid BCS$

$A \rightarrow aA \mid C$

$B \rightarrow bbB \mid b$

$C \rightarrow cC \mid \lambda$

5. $G: S \rightarrow ABC \mid aBC$

$A \rightarrow aA \mid BC$

$B \rightarrow bB \mid \lambda$

$C \rightarrow cC \mid \lambda$

6. 证明引理 4.3.2

从练习 7 到练习 10, 构造不包含链规则的等价文法 G_c 。为每种文法产生的语言写出正则表达式。注意这些文法不包含 λ 规则。

7. $G: S \rightarrow AS \mid A$

$A \rightarrow aA \mid bB \mid C$

$B \rightarrow bB \mid b$

$C \rightarrow cC \mid B$

8. $G: S \rightarrow A \mid B \mid C$

$A \rightarrow aa \mid B$

$B \rightarrow bb \mid C$

$C \rightarrow cc \mid A$

9. $G: S \rightarrow A \mid C$

$A \rightarrow aA \mid a \mid B$

$B \rightarrow bB \mid b$

$C \rightarrow cC \mid c \mid B$

10. $G: S \rightarrow AB \mid C$

$A \rightarrow aA \mid B$

$B \rightarrow bB \mid C$

$C \rightarrow cC \mid a \mid A$

11. 消除练习 1 中文法 G_L 中的链规则。

12. 消除练习 4 中文法 G_L 中的链规则。

13. 证明算法 4.4.2 能够生成推导出终结字符串的变量的集合。

[138]

从练习 14 到练习 16, 构造没有无用符的等价文法。跟踪用来构造 G_T 和 G_L 的 TERM 和 REACH 集合描述这些文法产生的语言。

14. $G: S \rightarrow AA \mid CD \mid bB$

$A \rightarrow aA \mid a$
 $B \rightarrow bB \mid bC$
 $C \rightarrow cB$
 $D \rightarrow dD \mid d$

15. $G: S \rightarrow aA \mid BD$

$A \rightarrow aA \mid aAB \mid aD$
 $B \rightarrow aB \mid aC \mid BF$
 $C \rightarrow Bb \mid aAC \mid E$
 $D \rightarrow bD \mid bC \mid b$
 $E \rightarrow aB \mid bC$
 $F \rightarrow aF \mid aG \mid a$
 $G \rightarrow a \mid b$

16. $G: S \rightarrow ACH \mid BB$

$A \rightarrow aA \mid aF$
 $B \rightarrow CFH \mid b$
 $C \rightarrow aC \mid DH$
 $D \rightarrow aD \mid BD \mid Ca$
 $F \rightarrow bB \mid b$
 $H \rightarrow dH \mid d$

139

17. 证明下面文法的所有字符都是有用的:

$G: S \rightarrow A \mid CB$
 $A \rightarrow C \mid D$
 $B \rightarrow bB \mid b$
 $C \rightarrow cC \mid c$
 $D \rightarrow dD \mid d$

从 G 中去掉链规则, 构造它的等价文法 G_c 。证明 G_c 包含无用符。

18. 把下面的文法转化成乔姆斯基范式:

$G: S \rightarrow aA \mid ABa$
 $A \rightarrow AA \mid a$
 $B \rightarrow AbB \mid bb$

G 已经满足关于开始符 S 、 λ 规则、无用符和链规则的条件了。

19. 把下面的文法转化成乔姆斯基范式。

$G: S \rightarrow aAbB \mid ABC \mid a$
 $A \rightarrow aA \mid a$
 $B \rightarrow bBcC \mid b$
 $C \rightarrow abc$

G 已经满足关于开始符 S 、 λ 规则、无用符和链规则的条件了。

20. 把练习 9 的结果转化成乔姆斯基范式。

21. 把练习 11 的结果转化成乔姆斯基范式。

22. 把练习 12 的结果转化成乔姆斯基范式。

23. 把下面的文法转化成乔姆斯基范式:

$G: S \rightarrow A \mid ABa \mid AbA$
 $A \rightarrow Aa \mid \lambda$
 $B \rightarrow Bb \mid BC$
 $C \rightarrow CB \mid CA \mid bB$

*24. 已知 G 是乔姆斯基范式的文法。

a) $L(G)$ 中长度为 n 的字符串的推导的长度是多少?

b) $L(G)$ 中长度为 n 的字符串的推导树的最大深度是多少?

c) $L(G)$ 中长度为 n 的字符串的推导树的最小深度是多少?

25. 当输入字符串是 $abbb$ 和 $aabbb$ 时, 给出使用例 4.5.2 中的乔姆斯基范式文法的 CYK 算法产生的上三角矩阵。

26. 已知 G 是乔姆斯基范式文法, 如下

$$\begin{aligned} S &\rightarrow AX \mid AY \mid a \\ X &\rightarrow AX \mid a \\ Y &\rightarrow BY \mid a \\ A &\rightarrow a \\ B &\rightarrow b. \end{aligned}$$

给出使用该文法且输入串是 $baaa$ 和 $abaaa$ 时的 CYK 算法产生的上三角矩阵。

27. 已知 G 是下面的文法

$$\begin{aligned} G: S &\rightarrow A \mid B \\ A &\rightarrow aaB \mid Aab \mid Aba \\ B &\rightarrow bB \mid Bb \mid aba. \end{aligned}$$

a) 给出 $L(G)$ 的正则表达式。

b) 构造与 G 等价的不包含左递归规则的文法 G' 。

28. 构造不包含左递归规则且等价于下面文法的文法 G' :

$$\begin{aligned} G: S &\rightarrow A \mid C \\ A &\rightarrow AaB \mid AaC \mid B \mid a \\ B &\rightarrow Bb \mid Cb \\ C &\rightarrow cC \mid c. \end{aligned}$$

给出字符串 $aaccacb$ 在文法 G 和 G' 中的最左推导。

29. 构造不包含左递归规则且等价于下面文法的文法 G' :

$$\begin{aligned} G: S &\rightarrow A \mid B \\ A &\rightarrow AAA \mid a \mid B \\ B &\rightarrow BBB \mid b. \end{aligned}$$

30. 构造等价于下面文法的格立巴赫范式:

$$\begin{aligned} S &\rightarrow aAb \mid a \\ A &\rightarrow SS \mid b. \end{aligned}$$

31. 把乔姆斯基范式

$$\begin{aligned} S &\rightarrow BB \\ A &\rightarrow AA \mid a \\ B &\rightarrow AA \mid BA \mid b \end{aligned}$$

转化成格立巴赫范式。处理变量的顺序是 S, A, B 。

32. 把乔姆斯基范式

$$\begin{aligned} S &\rightarrow AB \mid BC \\ A &\rightarrow AB \mid a \\ B &\rightarrow AA \mid CB \mid b \\ C &\rightarrow a \mid b \end{aligned}$$

转化成格立巴赫范式。处理变量的顺序是 S, A, B, C 。

33. 把乔姆斯基范式

$$\begin{aligned} S &\rightarrow BA \mid AB \mid \lambda \\ A &\rightarrow BB \mid AA \mid a \\ B &\rightarrow AA \mid b \end{aligned}$$

转化成格立巴赫范式。处理变量的顺序是 S, A, B 。

34. 把乔姆斯基范式

$$\begin{aligned}
 S &\rightarrow AB \\
 A &\rightarrow BB \mid CC \\
 B &\rightarrow AD \mid CA \\
 C &\rightarrow a \\
 D &\rightarrow b
 \end{aligned}$$

转化成格立巴赫范式。处理变量的顺序是 S, A, B, C, D 。

*35. 证明每个上下文无关语言都是使用具有下面形式的规则的文法生成的：

- i) $S \rightarrow \lambda$
- ii) $A \rightarrow a$
- iii) $A \rightarrow aB$, 或者
- iv) $A \rightarrow aBC$

其中, $A \in V, B, C \in V - \{S\}$ 并且 $a \in \Sigma$ 。

参考文献注释

消除 λ 规则和链规则的构造过程参见 Bar-Hillel、Perles 和 Shamir [1961]。乔姆斯基范式是乔姆斯基 [1959] 提出的。CYK 算法是以 J. Cocke、D. Younger [1967] 以及独立提出这种确定可推导性技术的 T. Kasami 的名字命名的。这个算法的变形可以用于解决任意上下文无关文法的成员问题, 而不需要把它转化成乔姆斯基范式。

格立巴赫范式是 Greibach [1965] 提出的。格立巴赫范式的另一种转换限制了结果文法中的规则的数目的增长, 详见 Blum 和 Koch [1999]。格立巴赫范式的定义有几种变形。通用的公式表示要求终结符要出现在字符串的第一个位置, 但是允许字符串的剩下部分包含变量和终结符。双格立巴赫范式, Engelfriet [1992], 要求规则右侧的最左和最右符号都是终结符。

规则满足练习 35 的条件文法可以称作是 2-范式。产生整个上下文无关语言的集合的 2-范式文法证明参见 Hopcroft、Ullman [1979] 和 Harrison [1978]。Harrison [1978] 给出了上下文无关文法的另一种范式。

第5章 有限自动机

本章我们将介绍被称为有限状态自动机的一类抽象计算装置。有限状态自动机是用来判定一个字符串是否满足一组条件，或者是否匹配一个规定的模式。有限状态自动机和许多机械装置具有一些共同的性质：它们对输入进行处理并产生输出。一台自动售货机将硬币作为输入，并返回食物或者饮料作为输出。一个号码锁期望一串数字，如果输入了正确的数字串将会打开锁。有限状态机的输入是一个字符串，运算的结果指出有限状态自动机是否接收该串。有限状态自动机接收的串的集合构成了自动机的语言。

前面的机器的例子展示机械计算的一个性质：确定性。当投放适当数量的钱到自动售货机中时，如果任何事情都不发生我们将非常担心。同样，我们期望一个输入的组合可以打开锁，而其他的序列不能打开。最初，我们要求有限状态自动机具有确定性。之后我们将放宽这个条件，来检查非确定性对有限状态计算的能力的影响。

5.1 一个有限状态自动机

一个机器的形式定义与该机器操作中涉及的硬件无关，但是却与机器处理输入时进行的内部操作的描述相关。一个自动售货机可能由若干个控制杆、带制动栓的号码锁，以及一个由微处理器控制的电子输入系统构成，但是不管哪种机器，它们都接收输入，并产生肯定的或否定的响应。什么描述能够包含这些看起来是不同类型的机械计算的功能？ [145]

采用一个简单的可以在很多街角看到的报纸自动售卖机来描述一个有限状态机器的各个组件。机器的输入可以是5美分、10美分和25美分的硬币。当插入了30美分，机器的盖子打开，并可以拿走一份报纸。如果硬币的总数超过了30美分，机器优雅地接收多给的钱，不找回零钱。

大街角落处的报纸售卖机没有记忆，至少不具有我们通常想象中的计算机器中的存储器。但是机器“知道”，如果已经插入了25美分，当再插入5美分的时候，机器的盖子将会打开。这些知识要求机器在接收并处理完输入的时候，可以改变自身的内部状态。

机器状态表示正在进行的计算的情况。下面的7个状态之间的交互可以描述自动售卖机的内部操作。用楷体表示出来的状态的名字指明了朝着打开盖子目标进展的过程。

- 需要30美分：没有插入任何硬币的机器状态
- 需要25美分：输入1个5美分硬币后的状态
- 需要20美分：输入2个5美分或者1个10美分硬币后的状态
- 需要15美分：输入3个5美分或者1个10美分和1个5美分硬币后的状态
- 需要10美分：输入4个5美分或者1个10美分和2个5美分或者2个10美分后的状态
- 需要5美分：输入1个25美分或者5个5美分或者2个10美分和1个5美分或者1个10美分和3个5美分后的状态
- 需要0美分：表示至少输入了30美分的状态

插入一个硬币导致机器改变状态。当30或更多美分输入后，进入需要0美分状态，并且锁打开。这个状态称为接收状态，因为它表明输入正确。

设计自动机的时候，必须用符号表示每一个构件。抽象机器的输入是一串符号而不是一串硬币。通常采用一个带标记的有向图，称为状态图（State Diagram），来表示机器的内部转换。状态图的节点表示上面所描述的状态。在状态图中，简单地用 m 表示需要 m 美分的节点。计算开始时机器的状态为 \times 。报纸售卖机的初始状态为节点30。

[146]

弧上具有标记 n 、 d 或者 q ，分别表示输入的是 5 美分、10 美分或者 25 美分的硬币。从节点 x 到节点 y 的标记为 v 的弧表示，当机器在状态 x 时处理输入 v 将使机器进入状态 y 。图 5-1 给出了报纸售卖机的状态图。从节点 15 到节点 5 的标记为 d 的弧表示了，机器在处理了前面的 15 美分的之后再输入 10 美分时的机器状态的改变。节点 0 到自身、长度为 1 的弧表示当总数超过 30 美分后，任何的输入都将使锁保持打开状态。

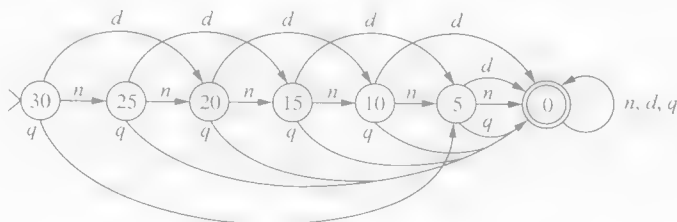


图 5-1 报纸售卖机的状态图

机器的输入由 $\{n, d, q\}$ 中的串组成。处理输入串的过程中进入的状态序列可以通过跟踪状态图中的弧获得。计算开始时，机器在初始状态。经过标记为第一个输入符号的弧，得到了机器状态的子序列。处理输入串中的下一个符号时，在当前的状态经过适当的弧到达前面弧的目标节点。重复这个过程直到输入串全部被处理完。如果计算终止在接收状态，则接收输入串。串 $dndn$ 能够被自动售卖机接收，而串 $nndn$ 不被接收，因为计算终止在状态 5。

5.2 确定型有限自动机

自动售卖机的分析需要将设计的基本原理从实现细节之中分离出来。这种独立于实现的描述通常叫做抽象机器。下面将介绍一类抽象机器，它们的计算可以用来决定是否接收输入串。

定义 5.2.1 确定型有限自动机 (DFA) 是一个五元组 $M = (Q, \Sigma, \delta, q_0, F)$ ，其中 Q 是一个有限状态集合； Σ 是一个有限字母表； $q_0 \in Q$ ， q_0 称为初始状态； F 是 Q 的子集，称为终结状态集或者接收状态集； δ 是一个从 $Q \times \Sigma$ 到 Q 的全函数，称为转换函数。

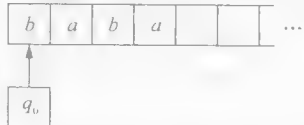
我们把一个确定型有限自动机看作一个抽象机器。为了揭示确定型有限自动机的机器本质，我们可以使用许多熟悉的计算机中的构件来描述 DFA 的操作。一个自动机包含五个组件：一个内部寄存器，一组寄存器值，一个带，一个带读取器和一个指令集。

[147]

我们通常用 DFA 的状态来表示机器的内部情形，记为 $q_0, q_1, q_2, \dots, q_n$ 。机器的寄存器，也称为有限控制器，包含一个状态值。计算开始时，寄存器的值为 DFA 的初始状态 q_0 。

DFA 的输入是字母表 Σ 中元素的一个有限序列。带上存储输入，直到计算使用这些输入。带被分成了许多的方格，每一个方格能够容纳字母表中的一个元素。由于输入串的长度没有上限，所以带也必须没有长度的限制。自动机一次计算的输入放置在带的一个初始段之中。

带头读取输入带的一个方格。机器的主体包含带头和寄存器两个部分。将主体放在正在扫描的带方格下面，这样就指明了带头的位置。寄存器的值表示自动机当前的状态。输入为 $baba$ 的计算的初始配置描述如下：



自动机的计算包括执行一序列的指令。执行一条指令将改变机器的状态并将带头向右移动一个方格。指令集从 DFA 的转换函数中获得。机器状态和扫描到的符号决定将要执行的指令。自动机在状态 q_i

下扫描到 a 的动作是将状态设置为 $\delta(q_i, a)$ 。因为 δ 是一个全函数，每一个状态和输入符号的组合对应惟一的指令，所以确定型有限状态自动机是确定的。

自动机计算的目的是确定输入串是否能够被自动机接收。计算开始时，带头扫描带的最左边的方格，此时寄存器之中含有状态 q_0 。根据状态和扫描到的符号选择指令。之后，机器改变为指令所给出的状态，带头右移。每个指令周期中机器的转换如图 5-2 所示。指令周期重复直到带头扫描到空白方

格,此时计算终止。如果计算终止在接收状态,那么接收输入串;否则,不接收输入串。图 5-2 展示了接收串 aba 的过程。

定义 5.2.2 $M = (Q, \Sigma, \delta, q_0, F)$ 是一个 DFA。 M 的语言 (Language), 记为 $L(M)$, 是 Σ^* 上能够被 M 接收的串的集合。

DFA 可以被看作是一个语言的接收器;简单地说,机器的语言就是能够被机器的计算接收的串的集合。图 5-2 中的机器的语言是 $\{a, b\}$ 上的所有以 a 结尾的字符串的集合。

DFA 是一个从左到右处理输入的只读机器;一旦输入字符被读入,它将不再影响计算。在计算的任何时候,计算的结果只依赖于当前的状态和未处理的输入。这种组合叫做机器格局 (Machine Configuration), 表示为有序对 $[q_i, w]$, 其中 q_i 是当前状态, $w \in \Sigma^*$ 是未处理的输入。DFA 的一个指令周期将一个机器格局转换到另一个机器格局。记号 $[q_i, aw] \vdash [q_j, w]$ 表示自动机 M 在格局 $[q_i, aw]$ 时,可以通过执行自动机 M 的一个指令周期而获得格局 $[q_j, w]$ 。记号 \vdash , 读做“产生”, 定义了一个从 $Q \times \Sigma^*$ 到 $Q \times \Sigma^*$ 的函数, 它可以用来跟踪 DFA 的计算。如果没有歧义我们将省略 M 。

定义 5.2.3 $Q \times \Sigma^*$ 上的函数 \vdash 定义如下。对任意的 $a \in \Sigma$, $w \in \Sigma^*$ 有

$$[q_i, aw] \vdash [\delta(q_i, a), w]$$

其中, δ 是 DFAM 的转换函数。

记号 $[q_i, u] \vdash^* [q_j, v]$ 表示从格局 $[q_i, u]$ 可以通过零步或者多步转换获得格局 $[q_j, v]$ 。

例 5.2.1 下面定义的 DFA M 可以接收 $\{a, b\}$ 上的所有含有 bb 子串的字符串的集合, 即, $L(M) = (a \cup b)^* bb (a \cup b)^*$ 。 M 的状态和字母表为:

$$\begin{aligned} M: Q &= \{q_0, q_1, q_2\} \\ \Sigma &= \{a, b\} \\ F &= \{q_2\}. \end{aligned}$$

转换函数 δ 以二维表格的形式给出, 叫做转换表。垂直方向是状态, 水平方向是字母表。自动机在状态 q_i 时输入 a 的动作可以通过查找 q_i 行、 a 列处的内容决定。

δ	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

下面采用函数 \vdash 跟踪 M 对输入串 $abba$ 和 $abab$ 的计算。

$[q_0, abba]$	$[q_0, abab]$
$\vdash [q_0, bba]$	$\vdash [q_0, bab]$
$\vdash [q_1, ba]$	$\vdash [q_1, ab]$
$\vdash [q_2, a]$	$\vdash [q_0, b]$
$\vdash [q_2, \lambda]$	$\vdash [q_1, \lambda]$
接收	拒绝

因为计算终止在状态 q_2 , 所以串 $abba$ 被接收。

例 5.2.2 前一节的报纸售卖机可以表示成一个 DFA, 它的状态、字母表和转换函数如下。开始状态在 30。

$$\begin{aligned} M: Q &= \{q_0, q_1\} & \delta(q_0, a) &= q_1 \\ \Sigma &= \{a, b\} & \delta(q_0, b) &= q_0 \\ F &= \{q_1\} & \delta(q_1, a) &= q_1 \\ & & \delta(q_1, b) &= q_0 \end{aligned}$$

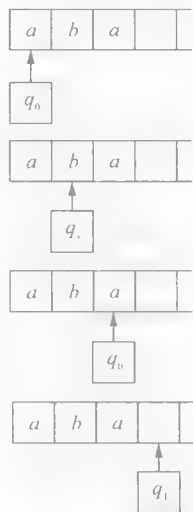


图 5-2 DFA 中的计算

$$Q = \{0, 5, 10, 15, 20, 25, 30\}$$

$$\Sigma = \{n, d, q\}$$

$$F = \{0\}$$

δ	n	d	q
0	0	0	0
5	0	0	0
10	5	0	0
15	10	5	0
20	15	10	0
25	20	15	0
30	25	20	5

[150]

售货机接收的语言是所有表示的和超过 30 美分的串。你能够构造一个定义该机器的语言的正则表达式吗? \square

转换函数详细说明了机器在任何给定的状态和字母表中的元素时的动作。这个函数可以扩展成 $\hat{\delta}$ ，它的输入包括一个状态和字母表上的一个串。函数 δ 可以通过递归地将范围从 Σ 中的元素扩展到任意长度的串来构造。

定义 5.2.4 DFA 转换函数 δ 的扩展转换函数 (extended transition function) $\hat{\delta}$ 是从 $Q \times \Sigma^*$ 到 Q 的函数。 $\hat{\delta}$ 的值由输入串的长度递归地定义。

i) 归纳基础: $\text{length}(w) = 0$, 那么 $w = \lambda$, 并且 $\hat{\delta}(q_i, \lambda) = q_i$,

$\text{length}(w) = 1$, 那么对所有 $a \in \Sigma$ 有 $w = a$, 并且 $\hat{\delta}(q_i, a) = \delta(q_i, a)$

ii) 递归步骤: 设 w 是长度为 $n > 1$ 的串, 那么 $w = ua$, 并且 $\hat{\delta}(q_i, ua) = \delta(\hat{\delta}(q_i, u), a)$

机器在状态 q_i 输入串 w 的计算将停止于状态 $\delta(q_i, w)$ 。函数 $\delta(q_i, w)$ 的价值在于, 它模拟了重复应用处理输入串 w 时所需要的转换函数的过程。如果 $\delta(q_i, w) \in F$, 那么串 w 被接收。使用这个记号, DFA M 的语言可以表示为 $L(M) = \{w \mid \hat{\delta}(q_0, w) \in F\}$ 。

5.3 状态图和例子

DFA 的状态图是一个带标记的有向图, 图的节点表示机器的状态, 图中的弧是通过转换函数获得的。图 5-1 是报纸售货机 DFA 的状态图。由于图的直观性, 我们经常采用状态图而不是集合和转换函数的形式来表示 DFA 的形式定义。

定义 5.3.1 DFA $M = (Q, \Sigma, \delta, q_0, F)$, 是一个有向图 G , 它满足如下的条件:

i) G 的节点是 Q 的元素。

ii) G 中弧上的标记是 Σ 中的元素。

iii) q_0 是开始节点, 表示为 \odot 。

iv) F 是接收节点的集合, 每一个接收节点表示为 \odot 。

v) 如果 $\delta(q_i, a) = q_j$, 则存在一条从 q_i 到 q_j 的标记为 a 的弧。

vi) 对每一个节点 q_i 和 $a \in \Sigma$, 存在惟一的一条以 q_i 为起点的标记为 a 的弧。

DFA 的一个转换表示为状态图中的一条弧。在 DFA 相应的状态图中, 通过追踪 DFA 的计算可以构造出一条以节点 q_0 为起点且弧上的标记为输入串的路径。设 p_w 是一条以 q_0 为起点标记为 w 的路径, q_w 是该路径的终止节点。定理 5.3.2 将证明对每一个串 $w \in \Sigma^*$ 存在惟一的一条这种路径。此外, q_w 是 DFA 处理 w 结束时的状态。

定理 5.3.2 设 DFA $M = (Q, \Sigma, \delta, q_0, F)$, $w \in \Sigma^*$ 。那么 w 决定 M 的状态图上惟一的一条路径 p_w 并且 $\hat{\delta}(q_0, w) = q_w$ 。

证明: 通过对串的长度进行归纳来证明该定理。如果 w 的长度为 0, 那么 $\delta(q_0, \lambda) = q_0$ 。对应的路径是一条空路径, 开始和终止节点都是 q_0 。

假设结论对所有长度小于或者等于 n 的串都成立。设 $w = ua$ 是长度为 $n + 1$ 的串。由归纳假设可知, 存在惟一的路径 p_u , 该路径上的标记为 u , 并且 $\hat{\delta}(q_0, u) = q_u$ 。路径 p_w 可由沿着 q_u

[151]

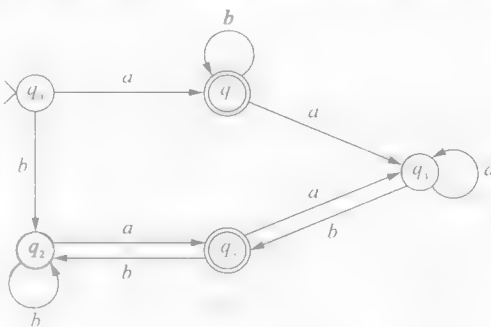
标记为 a 的弧构造。这是一条从 q_0 出发的标记为 w 的惟一的路径, 因为 p_u 是标记为 u 的惟一路径, 并且从 q_u 出发只有一条标记为 a 的弧。路径 p_u 的终止状态由转换 $\delta(q_u, a)$ 决定。根据扩展转换函数的定义可得 $\hat{\delta}(q_0, w) = \delta(\hat{\delta}(q_0, u), a)$ 。因为 $\hat{\delta}(q_0, u) = q_u$, 所以 $q_w = \delta(q_u, a) = \delta(\hat{\delta}(q_0, u), a) = \hat{\delta}(q_0, w)$, 证毕。

DFA 的计算和状态图中的路径的等价性, 给了我们一个确定 DFA 的语言的启发式的方法。状态 q_i 接收的串就是从 q_0 到 q_i 的所有路径上的标记。我们可以将这个�过程分离成两部分:

- 首先, 找到从 q_0 第一次到达 q_i 的所有串的正则表达式 u_1, \dots, u_n 。
- 找到所有离开 q_i 后又回到 q_i 的路径的正则表达式 v_1, \dots, v_m 。

则状态 q_i 接收的串为 $(u_1 \cup \dots \cup u_n)(v_1 \cup \dots \cup v_m)^*$ 。

考虑 DFA M (如右图) 的语言由所有从 q_0 出发到达 q_1 或者 q_3 的路径的标记组成。采用前面的启发式的描述, 到每一个接收状态的串是:



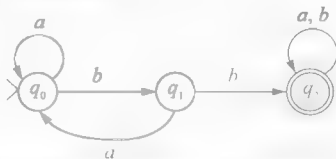
[152]

状态	到 q_i 的路径	从 q_i 到 q_i 的简单回路	接受的串
q_1	a	b	ab^*
q_3	ab^*aa^*b, bb^*a	bb^*a, aa^*b	$(ab^*aa^* \cup ba)(ab \cup ba)^*$

因此, $L(M) = ab^* \cup (ab^*aa^*b \cup bb^*a)(aa^*b \cup bb^*a)^*$ 。当我们确定了有限状态计算的其他性质之后, 将给出一个能够自动产生有限自动机语言的正则表达式的算法。

接下来我们将通过具体例子来帮助设计检查具有一定模式的串的自动机。我们将要考虑的情形包括出现次数以及指定子串的相对位置。此外, 我们还要确定接收语言 L 和 L 的补的 DFA 之间的关系。

例 5.3.1 例 5.2.1 中的 DFA 的状态图如下:

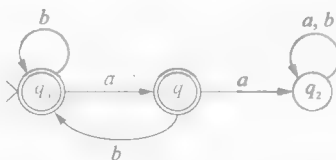


这些状态用来记录处理到的连续的 b 的个数。当遇到一个子串 bb 时, 自动机将进入状态 q_2 。一旦自动机进入状态 q_2 , 接下来处理的输入将不会改变自动机的状态。这个 DFA 对输入 $ababb$ 的计算和状态图中相应的路径如右表所示。串 $ababb$ 能够被接收, 因为计算的终止状态是接收状态 q_2 , 它也是具有标记 $ababb$ 的路径的终止状态。

计算	路径
$[q_0, ababb]$	q_0 ,
$\vdash q_0, babbb$	q_0 ,
$\vdash q_1, abb$	q_1 ,
$\vdash q_0, bbb$	q_0 ,
$\vdash q_1, b$	q_1 ,
$\vdash [q_2, \lambda]$	q_2

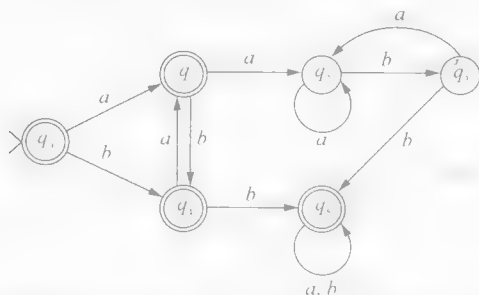
[153]

例 5.3.2 DFA



接收 $(b \cup ab)^*(a \cup \lambda)$, 即 $\{a, b\}$ 上的所有不包含子串 aa 的串的集合。

例 5.3.3 接收 $\{a, b\}$ 上含有子串 bb 或者不含有子串 aa 的串的 DFA 的描述如下。该语言是前面两个例子的语言的并集。



□

和接收两个语言的并集的机器相比较,接收含子串 bb 或者不含子串 aa 的机器的状态图看起来要简单些。看上去不存在一种直接的通过组合两个已知 DFA 的状态图,来构造一个所要求的复杂的机器的方法。

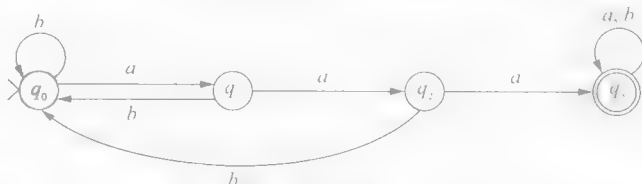
下面的几个例子将为设计 DFA 提供启发。第一步是获得一个 DFA 状态的解释,状态的解释描述了当自动机处于该状态时已经被处理的串的性质。相关的性质由接收一个串所需要的条件来决定。

例 5.3.4 一个接收 $\{a, b\}$ 上的含有子串 aaa 的串的 DFA 的一次成功计算必须处理连续的 3 个 a 。需要 4 个状态来记录计算检查串 aaa 的情形。状态的解释和名字见右表。

在处理第一个符号之前,没有处理查找 aaa 的过程。因此,这个条件表示了初始状态。

一旦状态被标识,那么通常很容易决定适当的转换。当计算在状态 q_1 处理了一个 a 且最后读取的两个字符是 aa 时,则进入 q_2 。另一方面,如果在状态 q_1 读入 b ,那么结果串表示前面没有处理过的 aaa ,计算进入 q_0 。根据相似的策略,可以为 DFA 的所有状态决定转换。

状态	解释
q_0 :	未处理 aaa
q_1 :	处理的最后一个字符是 a
q_2 :	处理的最后两个字符是 aa
q_3 :	在串中找到 aaa

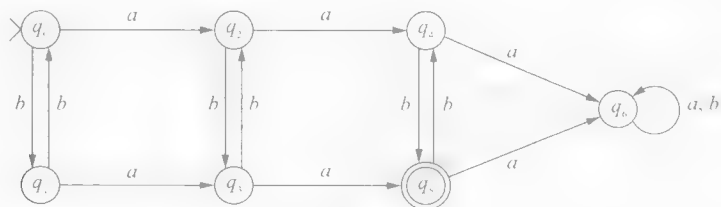


一旦处理 aaa ,则计算进入 q_3 ,读入剩余的串,并接受输入。 □

例 5.3.5 构造一个只接收含有两个 a 和奇数个 b 的串的自动机,需要检查两个条件: a 的个数和 b 的个数的奇偶性。需要 7 个状态来存储这个串的信息。状态的解释描述了读入 a 的个数和处理的 b 的个数的奇偶性。

在计算开始时,没有 a 和 b 被处理,这成为开始状态的条件。接收这个语言的 DFA 如下:

状态	解释
q_0 :	无 a , 偶数个 b
q_1 :	无 a , 奇数个 b
q_2 :	1 个 a , 偶数个 b
q_3 :	1 个 a , 奇数个 b
q_4 :	2 个 a , 偶数个 b
q_5 :	2 个 a , 奇数个 b
q_6 :	多于 2 个 a



水平弧记录输入串中 a 的个数,垂直的几对弧记录 b 的奇偶性。 q_5 是接收状态,因为它表示了语言中的串所需要的条件。 □

例 5.3.6 设 $\Sigma = \{0, 1, 2, 3\}$, Σ^* 中的一个串是 Σ 中数的一个序列。DFA 决定输入串中各位数字的和能否被 4 整除。例如, 串 12302 和 0130 能够被 M 接收, 0111 将被 M 拒绝。状态表示了已经处理的输入的和模 4 的值。

DFA 的定义只允许有两种可能的输出, 接收或者拒绝。输出可以扩展为每个状态关联一个值。计算的结果是与计算终止状态关联的值。这种类型的机器叫做 Moore 机, 因为这种类型的有限状态计算是由 E. F. Moore 引入。将值 i 与状态 $i \bmod 4$ 相关联, 例 5.3.6 中的机器将扮演一个模加法器。

例 5.3.1、例 5.3.2 和例 5.3.3 的机器的状态图显示, 不存在一种简单的方法, 采用这种方法我们可以从两个分别接收两种语言的 DFA 构造一个接收这两种语言的并的 DFA。下面的两个例子将展示接收补集的机器的不同情况。一个 DFA 的状态图可以很容易地转换为另一个接收并且只接收该 DFA 拒绝接收的串的自动机的状态图。

例 5.3.7 DFA M 接收 $\{a, b\}$ 上所有包含偶数个 a 和奇数个 b 的串。

在计算的任何一步, 根据输入符号个数的奇偶性有四种可能的组合: (1) 偶数个 a , 偶数个 b , (2) 偶数个 a , 奇数个 b , (3) 奇数个 a , 偶数个 b , (4) 奇数个 a , 奇数个 b 。可以用一个有序对表示这四种状态, 有序对的第一部分表示 a 的奇偶性, 第二部分表示 b 的奇偶性。每处理一个符号都将改变其中的一个部分, 由此可以设计出适当的转换。

例 5.3.8 设 M 是例 5.3.7 中构造的 DFA。DFA M' 被构造为接收 $\{a, b\}$ 上的所有不含奇数个 a 和偶数个 b 的串。换言之, 即 $L(M') = \{a, b\}^* - L(M)$ 。任何被 M 拒绝的串将被 M' 接收, 反之亦然。M' 的状态图可以通过交换 M 的接收和非接收状态获得。

前面的例子说明了接收互补串的集合的 DFA 之间的关系。下面的定理对该关系进行了形式化。

定理 5.3.3 设 DFA $M = (Q, \Sigma, \delta, q_0, F)$, 那么 $M' = (Q, \Sigma, \delta, q_0, Q - F)$ 是一个 DFA, 并且 $L(M') = \Sigma^* - L(M)$ 。

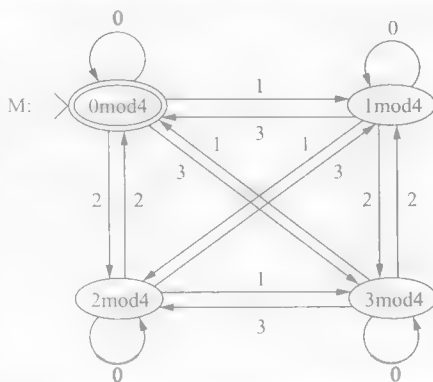
证明: 设 $w \in \Sigma^*$, $\hat{\delta}$ 是由 δ 构造的扩展转换函数。对每一个 $w \in L(M)$, 有 $\hat{\delta}(q_0, w) \in F$ 。所以, $w \notin L(M')$ 。相反, 如果 $w \notin L(M)$, 则 $\hat{\delta}(q_0, w) \in Q - F$ 并且 $w \in L(M')$ 。

由定义, DFA 必须处理完所有的输入, 甚至在结果已经确定的情况下仍需如此。例 5.3.9 展示了另一种确定性, 有时称为不完全确定; 每一个格局至多具有一个动作。该自动机的转换是由从 $Q \times \Sigma$ 到 Q 上的一个部分函数定义。一旦能够决定串是否被接收, 计算就停止, 在处理完整个输入前就终止的计算将拒绝接收输入。

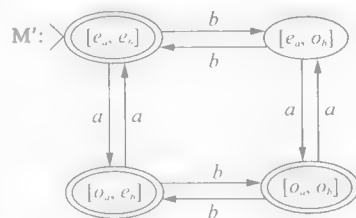
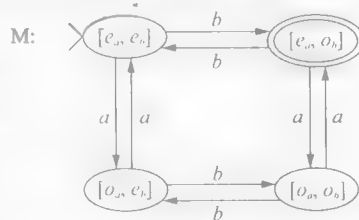
例 5.3.9 下面的状态图定义了一个不完全指定的 DFA, 它接收 $(ab)^*c$ 。一旦输入与要求的模式不同, 计算就终止。

输入 $abcc$ 的计算将被拒绝, 因为机器在状态 q_2 下不能处理最后的一个字符 c 。

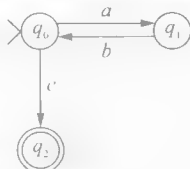
接收相同语言的两个自动机称为等价的。一个不完全指定的 DFA 可以容易地转换为一个等价的 DFA。转换需要一个额外的非接收状态的“错误”状态。



156



157

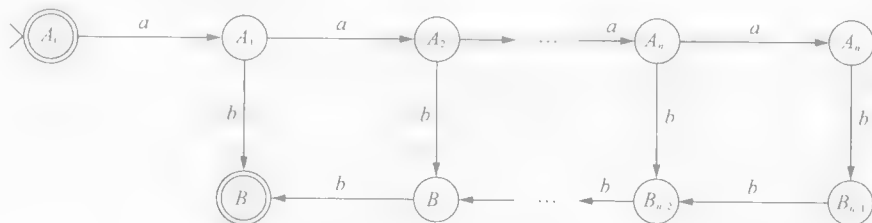
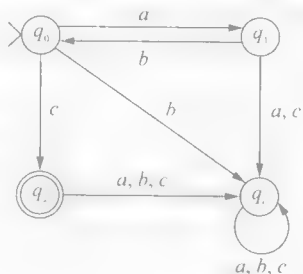


[158]

一旦不完全指定 DFA 达到一个没有动作的格局, 则 DFA 将进入这个错误状态。在错误状态上, DFA 读取所有的剩下的串, 并终止。

例 5.3.10 DFA 与例 5.3.9 中的不完全指定 DFA 接收相同的语言。状态 q_e 是一个错误状态, 它保证了整个串被处理。□

例 5.3.11 对每一个确定的 n , 下面的状态图定义的不完全指定 DFA 接收语言 $\{a^i b^i \mid i \leq n\}$ 。状态 A_k 记录 a 的个数, 同时状态 B_k 保证了 b 和 a 的个数相同。但是它不能扩展为接收 $\{a^i b^i \mid i \geq 0\}$ 的 DFA, 因为这需要无限个状态。下一章我们将说明语言 $\{a^i b^i \mid i \geq 0\}$ 不能被任何的有限自动机接收。



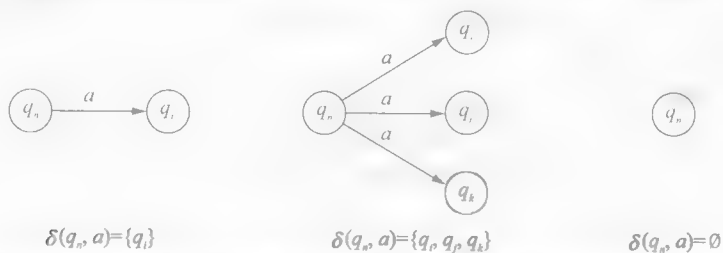
5.4 非确定型有限自动机

□

[159]

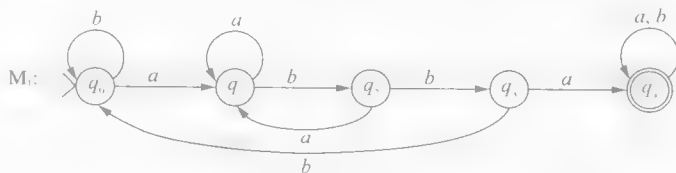
现在我们改变对机器的定义以允许非确定型的计算。在一个非确定型的自动机中, 任意一个给定的机器格局下有多条指令可以被执行。这个性质对于自动机看起来可能不是很自然, 但是非确定性通常使得设计语言接收器更加的容易。

一个非确定型有限自动机 (NFA) 的转换同 DFA 的转换具有相同的作用: 根据当前状态和扫描到的符号改变机器状态。对一个给定的机器格局, 转换函数必须指定机器可能进入的所有状态。只需要让一个状态的集合作为转换函数的值就可达到此目的。采用状态图的图形表示可用来说明非确定型计算的不同之处。任意有限数目的转换可以由给定的状态 q_n 和符号 a 确定。非确定型转换函数的值在下面相应的图表中给出。



由于非确定型计算和相应的确定型计算有很大的不同, 因此我们的介绍将从一个非确定型自动机的例子开始, 这个例子展示了两种计算类型的基本区别。此外, 我们将用例子介绍非确定型计算的特征, 并给出非确定型计算的直观解释。

考虑 DFA M_1



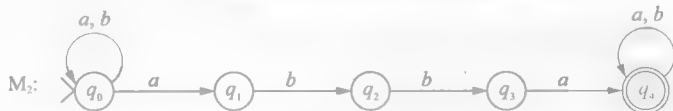
它接收语言 $(a \cup b)^* abba (a \cup b)^*$, 即在 a, b 上的包含子串 $abba$ 的所有串。状态 q_0, q_1, q_2 和 q_3 记录了处理子串 $abba$ 的过程。机器的状态如右表所示。当处理了 $abba$ 后, 便进入状态 q_4 , 即读入余下的串并接收输入。

当发现当前子串中不含有要求的形式时, 确定型计算在状态 q_0, q_1, q_2 和 q_3 必须“倒退”。当机器在状态 q_3 时扫描到 b , 则进入状态 q_0 , 因为最后处理的四个字符是 $abbb$, 当前的格局表示没有向前处理查找 $abba$ 的过程。

状 态	解 释
q_0 :	未处理串 $abba$
q_1 :	最后处理的一个字符是 a
q_2 :	最后处理的两个字符是 ab
q_3 :	最后处理的三个字符是 abb

[160]

接收 $(a \cup b)^* abba (a \cup b)^*$ 的一个非确定型的方法如下面的自动机所示:



在状态 q_0 的 M_2 , 处理 a 时存在两种可能的转换: 一种是 M_2 继续位于状态 q_0 , 并读入余下的串; 另一种是进入状态序列 q_1, q_2 和 q_3 , 检查接下来的三个符号是否构成子串 $abba$ 。

首先要注意的是非确定型自动机对一个输入串有多个计算。例如, M_2 对输入串 $aabbbaa$ 有 5 个不同的计算。我们将采用 5.2 节之中引入的符号 \vdash 来跟踪计算:

$\vdash [q_0, aabbbaa]$	$\vdash [q_0, aabbbaa]$	$\vdash [q_0, aabbbaa]$	$\vdash [q_0, aabbbaa]$	$\vdash [q_0, aabbbaa]$
$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_1, abbaa]$
$\vdash [q_0, bbaa]$	$\vdash [q_0, bbaa]$	$\vdash [q_0, bbaa]$	$\vdash [q_1, bbaa]$	
$\vdash [q_0, baa]$	$\vdash [q_0, baa]$	$\vdash [q_0, baa]$	$\vdash [q_2, baa]$	
$\vdash [q_0, aa]$	$\vdash [q_0, aa]$	$\vdash [q_0, aa]$	$\vdash [q_3, aa]$	
$\vdash [q_0, a]$	$\vdash [q_0, a]$	$\vdash [q_1, a]$	$\vdash [q_4, a]$	
$\vdash [q_0, \lambda]$	$\vdash [q_1, \lambda]$		$\vdash [q_4, \lambda]$	

当一个串具有几个终止在接收状态的计算, 而其他的计算终止在拒绝状态时, 接收一个串是什么意思呢? 答案在于使用前面段落之中提到的词检查。NFA 是设计来用于检查一个状态是否得到了满足, 在这个例子中, 即输入串是否含有子串 $abba$ 。如果其中一个计算发现子串存在, 则条件满足, 串被接收。与不完全指定的 DFA 一样, 为了获得一个肯定的答案就必须读完整个输入串。概括起来, 如果至少存在 NFA 的一个计算满足下面的条件, 则输入串被 NFA 接收:

- i) 处理整个串, 并且
- ii) 停止在接收状态。

如果存在一个计算接收一个串, 那么这个串就在非确定型自动机的语言中; 是否存在其他的不接收该串的计算与接收该串无关。

[161]

设计非确定型自动机时, 通常采用一种“猜测与检查”的策略。 M_2 中从状态 q_0 到 q_1 的转换做了这样的猜测, 即读入的 a 是 $abba$ 子串中的第一个 a 。猜测之后, 计算继续进入状态 q_1, q_2 和 q_3 来检查猜测是否正确。如果猜测之后的符号是 bba , 那么串被接收。

如果输入串含有子串 $abba$, 其中的一个猜测使 M_2 在读到子串的初始 a 时进入状态 q_1 , 那么这个计算将接收这个串。此外, M_2 只有在已经处理到子串 $abba$ 时才会进入状态 q_4 。因此, M_2 的语言是 $(a \cup b)^* abba (a \cup b)^*$ 。需要注意的是, 接收计算可以是不惟一的; 如 M_2 上存在两个不同的计算接收串 $abbabba$ 。

如果这是你第一次遇到非确定型, 那可能会问: 执行这种计算的自动机的能力如何呢? DFA 可以在软件和硬件中很容易地实现, 类似的 NFA 的实现怎么样呢? 我们可以直观的把非确定型计算想象为一种多进程。当计算进入一个具有多个转换的机器格局时, 会为每一种选择产生一个新的进程。根

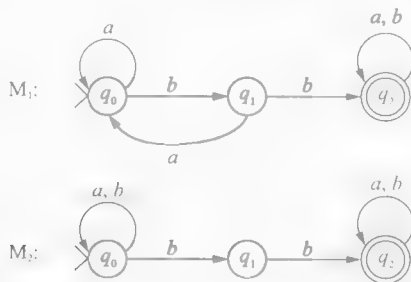
从图中我们很容易看出, 一个串当且仅当它以 bb 结尾时才能被 M 接收。

正如前面提到的, 一个 NFA 对一个输入串可以有多个计算。下面是对串 $ababb$ 的三个计算:

$[q_0, ababb]$	$[q_0, ababb]$	$[q_0, ababb]$
$\vdash [q_0, babb]$	$\vdash [q_0, babb]$	$\vdash [q_0, babb]$
$\vdash [q_0, abb]$	$\vdash [q_1, abb]$	$\vdash [q_0, abb]$
$\vdash [q_0, bb]$		$\vdash [q_0, bb]$
$\vdash [q_0, b]$		$\vdash [q_1, b]$
$\vdash [q_0, \lambda]$		$\vdash [q_2, \lambda]$

第二个计算执行了三条指令后终止, 因为机器在状态 q_1 扫描到 a 时没有指定的动作。第一个计算处理完整个串, 并终止在拒绝状态, 而最后一个计算终止在接收状态。第三个计算表明串 $ababb$ 在 M 的语言中。□

例 5.4.2 下面是接收 $(a \cup b)^* bb(a \cup b)^*$ 的有限自动机 M_1 和 M_2 的状态图。

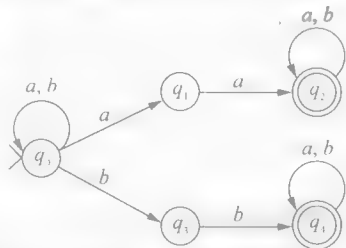


164

M_1 是例 5.3.1 中的 DFA。当遇到了第一个 bb 子串时, M_1 进入状态 q_2 并接收串。而 M_2 可以根据处理到的任意一个 bb 子串进入接收状态。□

例 5.4.3 我们可以把一个接收含有 bb 子串的机器和一个类似的接收含有 aa 子串的机器组合起来构造一个 NFA, 该 NFA 接收 $\{a, b\}$ 上的含有 aa 或者 bb 子串的所有串。

一条接收串的路径是: 在状态 q_0 读取输入, 直到遇到 aa 或者 bb 子串, 这时, 路径的分支根据相应的子串分别进入 q_1 或者 q_3 。图中存在三条不同的接收串 $abaaabb$ 的路径。□



通过使用非确定型所获得的弹性, 并不总是能够简化利用 M_1 和 M_2 来构造一个接收 $L(M_1) \cup L(M_2)$ 的自动机的过程。这一点可以从想要构造一个接收例 5.3.3 的 DFA 的语言的 NFA 看出来。

5.5 λ -转换

在确定型和非确定型自动机之中, 从一个状态到另一个状态的转换, 都是从处理输入字符开始的。现在需要把 NFA 的定义放宽, 从而允许不需要处理任何输入就进行状态转换。这种形式的转换称为 λ -转换 (λ -transition)。利用 λ -转换的非确定型自动机记做 NFA- λ 。

将 λ -转换引入到有限状态自动机之中, 是偏离 DFA 确定性计算的步骤。然而, 它们确实为设计接收复杂语言的自动机提供了有用的工具。

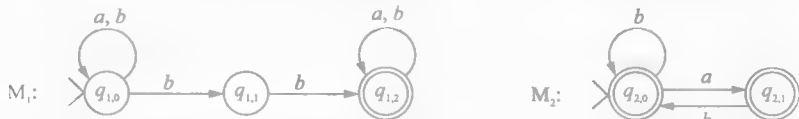
165

定义 5.5.1 带 λ -转换的非确定型有限自动机是一个五元组 $M = (Q, \Sigma, \delta, q_0, F)$, 其中 Q 、 δ 、 q_0 和 F 和 NFA 中的一样。转换函数是一个从 $Q \times (\Sigma \cup \{\lambda\})$ 到 $\mathcal{P}(Q)$ 的函数。

我们必须扩展计算终止的定义, 以包括下面的情况: 计算在处理完输入串后, 可能继续使用几个 λ -转换。我们应采用 NFA 之中使用的接收准则: 如果存在一个处理完整串并且停止在接收状态的计

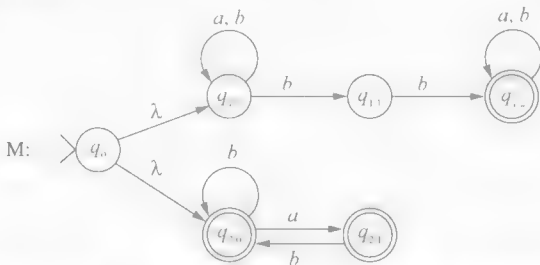
算,那么输入被接收 和前面的一样, $\text{NFA}-\lambda$ 的语言记做 $L(M)$ 。 $\text{NFA}-\lambda$ 的状态图可以根据定义 5.3.3 来构造,其中 λ -转换表示为带有 λ 标记的弧。

不需要处理输入符号就能在状态之间移动的能力,可以用来构造从简单到复杂的自动机。自动机 M_1 和 M_2 分别接收语言 $(a \cup b)^*bb(a \cup b)^*$ 和 $(b \cup ab)^*(a \cup \lambda)$ 。



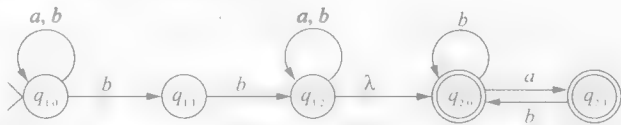
可以适当的组合 M_1 和 M_2 的状态图来构造复杂的机器。

例 5.5.1 $\text{NFA}-\lambda$ M 的语言为 $L(M_1) \cup L(M_2)$,



组合自动机 M 的计算经过一条 λ 弧进入 M_1 或者 M_2 的开始状态。如果存在一条 M_1 上的接收串的路径 p ,那么该串能够被 M 接收,路径是由从 q_0 到 $q_{1,0}$ 的 λ 弧和路径 p 构成的。因为在每一个计算的初始移动时都不处理输入字符,所以 M 的语言是 $L(M_1) \cup L(M_2)$ 。请将这种方法构造的机器的简单性和例 5.3.3 之中的确定型状态图进行比较。 □

例 5.5.2 一个接收 $L(M_1)L(M_2)$ 的 $\text{NFA}-\lambda$,即 M_1 和 M_2 接收的语言 L 的连接,它可以通过用一条 λ 弧将两个自动机连接起来的方法构造。

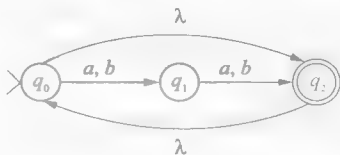


一个输入串被接收,当且仅当它由一个 $L(M_1)$ 中的串连接一个 $L(M_2)$ 中的串组成。无论何时如果输入串的前缀被 M_1 接收,则 λ -转换允许计算进入 M_2 。 □

例 5.5.3 我们将使用 λ -转换来构造一个 $\text{NFA}-\lambda$,它接收任何 $\{a, b\}$ 上的长度为偶数的串。我们从构造接收长度为 2 的串的自动机的状态图开始。



为了接收空串,需要再加一条从 q_0 到 q_1 的 λ -弧。任何长度为正偶数的串通过沿着从 q_2 到 q_0 的 λ -弧重复序列 q_0, q_1, q_2 而被接收。



例 5.5.1、例 5.5.2 和例 5.5.3 之中的构造可以概括为构造接收已知有限状态机接收的语言的并、连接和 Kleene 星闭包的自动机。第一步是将机器转化为等价的 $\text{NFA}-\lambda$ 。 □

引理 5.5.2 设 $\text{NFA-}\lambda M = (Q, \Sigma, \delta, q_0, F)$, 存在一个等价的 $\text{NFA-}\lambda M' = (Q \cup \{q'_0, q_f\}, \Sigma, \delta', q'_0, \{q_f\})$, 它满足如下的条件

- i) 开始状态 q'_0 的入度是 0。
- ii) 只有一个接收状态 q_f 。
- iii) 接收状态 q_f 的出度是 0。

167

证明: M' 的转换函数可以在 M 的转换函数上增加对新状态 q'_0 和 q_f 的 λ -转换获得:

$$\delta(q'_0, \lambda) = \{q_0\}$$

$$\delta(q_i, \lambda) = \{q_f\}, \text{ 对每个 } q_i \in F$$

从 q'_0 到 q_0 的 λ -转换允许计算在不影响输入的情况下, 对原始的自动机 M 进行处理。 M' 中接收一个输入串的计算和 M 是一样的, 只是在 M 的接收状态上增加一个到 M' 的接收状态的 λ -转换。 ■

如果一个自动机满足引理 5.5.2 的条件, 那么开始状态的所有作用是初始化计算, 并且一旦进入状态 q_f , 计算就终止。这样的一个自动机如下图所示:

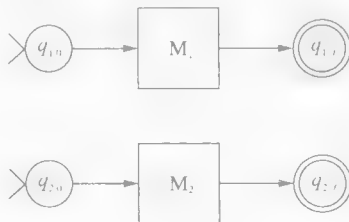


图中展示了一个具有三个不同组成部分的机器: 初始状态、机身和最终状态。这个可以比作一个在两端有连接器的有轨电车。事实上, 开始和最终状态的条件是用来允许它们扮演有限状态自动机的连接器的角色。

定理 5.5.3 设 M_1, M_2 是两个 $\text{NFA-}\lambda$, 存在接收 $L(M_1) \cup L(M_2)$ 、 $L(M_1)L(M_2)$ 和 $L(M_1)^*$ 的 $\text{NFA-}\lambda$ 。

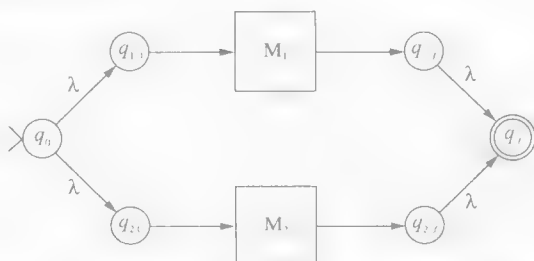
证明: 不失一般性地假设 M_1, M_2 满足引理 5.5.2 的条件。那么, 构造出来的接收语言 $L(M_1) \cup L(M_2)$ 、 $L(M_1)L(M_2)$ 和 $L(M_1)^*$ 的自动机也满足引理 5.5.2 的条件。

由于开始和最终状态的约束, M_1, M_2 可以描述如下:



168

接收语言 $L(M_1) \cup L(M_2)$ 的机器如下:



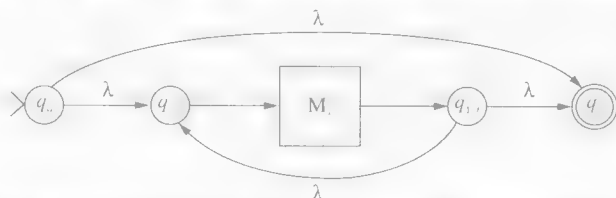
计算开始, 首先沿着 λ -弧进入 M_1 或 M_2 。如果串被其中的一个自动机接收, 那么可以经过 λ -弧进入复合的自动机的接收状态。可以把这个构造看作是建造一个同时运行 M_1 和 M_2 的机器。如果任何一个机器成功地处理完输入串, 那么输入被接收。

连接可以通过连续地操作各个组成自动机获得。复合的自动机的开始状态是 $q_{1,0}$, 接收状态是 $q_{2,f}$ 。将 M_1 的终止状态连接到 M_2 的开始状态, 这两个自动机就连连接在一起了。



当一个输入串的一个前缀被 M_1 接收时, 计算会继续进行并进入 M_2 。如果余下的串被 M_2 接收, 那么处理会终止在复合自动机的接收状态 $q_{2,f}$ 。

一个接收 $L(M_1)^*$ 的自动机必须能够循环穿过 M_1 若干次。从状态 $q_{1,i}$ 进入到状态 $q_{1,i}$ 的 λ -弧使其能够构成圈。为了接收空串, 需要另一条从状态 $q_{1,i}$ 到状态 $q_{1,i}$ 的 λ -弧。这些弧加到 M 上会产生:



[169] 在第6章中, 我们将用到这种形式的重复连接自动机, 来确定一个正则表达式所描述的语言和有限状态机器接收的语言的等价性。

5.6 去掉非确定性

在前面几节中, 我们已经介绍了三类有限自动机, 每一类都由它之前介绍的自动机产生。通过放宽对确定性的约束, 我们是否制造了一种能力更强的机器呢? 更准确地说, 是否存在一个能够被 NFA 接收但是不能够被 DFA 接收的语言呢? 我们将会说明并不存在这样的例子。此外, 还将给出一个 NFA- λ 转换为等价的 DFA 的算法。

DFA 和 NFA 中的状态转换都伴随着对一个输入符号的处理。为了将 NFA- λ 中的转换和对输入的处理联系起来, 我们构造了一个新的转换函数 t , 叫做输入转换函数, 它的值是在给定的状态下通过处理一个输入符号所能够进入的状态的集合。图 5-3 中的转换 $t(q_1, a)$ 的值为 $\{q_2, q_4, q_5, q_6\}$ 。状态 q_1 不在其中, 因为从 q_1 到状态 q_4 的转换不需要处理任何的输入符号。

路径	处理的字符串
q_1, q_2	a
q_1, q_2, q_3	a
q_1, q_4	λ
q_1, q_4, q_5	a
q_1, q_4, q_5, q_6	a

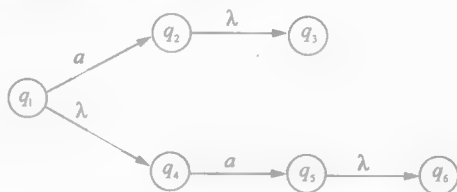


图 5-3 带 λ -转换的路径

直观地, 输入转换函数 $t(q_i, a)$ 的定义可以分为三个部分: 首先, 构造从状态 q_i 不需要处理输入符号就能够到达的状态的集合; 接着, 获得集合中所有的状态处理 a 之后所到达的状态的集合; 最后, 这些状态经过 λ -弧能够到达的状态的集合就构成了 $t(q_i, a)$ 。

函数 t 通过采用转换函数 δ 和状态图中标识为空串的路径来进行定义的。如果存在一条从 q_i 到 q_j 的标记为空串的路径, 则称状态 q_j 在 q_i 的 λ -闭包之中。

定义 5.6.1 状态 q_i 的 λ -闭包, 记做 $\lambda\text{-closure}(q_i)$, 递归定义如下:

- 基础步骤: $q_i \in \lambda\text{-closure}(q_i)$ 。
- 递归步骤: 设 q_j 是 $\lambda\text{-closure}(q_i)$ 之中的元素, 如果 $q_k \in \delta(q_j, \lambda)$, 则 $q_k \in \lambda\text{-closure}(q_i)$ 。
- 闭包: $q_j \in \lambda\text{-closure}(q_i)$, 当且仅当从 q_i 能够通过应用有限步的递归步骤获得 q_j 。

集合 $\lambda\text{-closure}(q_i)$ 可以根据算法 4.3.1 中的自顶向下的方法获得。它决定了在上下文无关的语法中的链。输入转换函数可用从状态的 λ -闭包和 NFA- λ 的转换函数中获得。

定义 5.6.2 NFA- λ M 的输入转换函数 t 是从 $Q \times \Sigma$ 到 $\mathcal{P}(Q)$ 的函数, 定义如下:

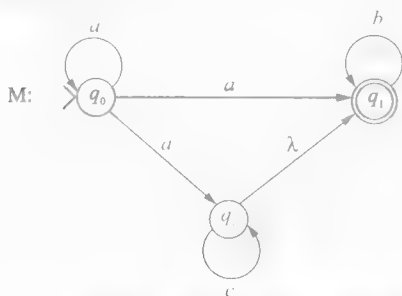
$$t(q_i, a) = \bigcup_{q_j \in \lambda\text{-closure}(q_i)} \lambda\text{-closure}(\delta(q_j, a)),$$

其中 δ 是 M 的转换函数。

[170]

输入转换函数和 NFA 的转换函数具有相同的形式。即, 输入转换函数是从 $Q \times \Sigma$ 到状态集合的函数。不带 λ -转换的 NFA 的输入转换函数 t 和自动机的转换函数 δ 相同。

例 5.6.1 转换函数 δ 和 NFA- λ 的输入转换函数 t 的转换表如下所示, M 是 NFA- λ 的状态图 M 接收的语言是: $a^+b^+c^+$ 。



δ	a	b	c	λ
q_0	$\{q_0, q_1, q_2\}$	\emptyset	\emptyset	\emptyset
q_1	\emptyset	$\{q_1\}$	\emptyset	\emptyset
q_2	\emptyset	\emptyset	$\{q_2\}$	$\{q_1\}$

t	a	b	c
q_0	$\{q_0, q_1, q_2\}$	\emptyset	\emptyset
q_1	\emptyset	$\{q_1\}$	\emptyset
q_2	\emptyset	$\{q_1\}$	$\{q_1, q_2\}$

□

NFA- λ 的输入转换函数可以用来构建一个等价的 DFA。非确定型自动机是否接收输入, 取决于是否存在能够处理完整串并停止在接收状态的计算。在 NFA- λ 状态图中, 可能存在几条表示输入串处理的路径, 而在 DFA 状态图中只存在唯一的路径。为了去掉非确定性, DFA 必须模拟同时探测 NFA- λ 中所有可能的计算。

算法 5.6.3 迭代地构造一个与 NFA- λ M 等价的确定型自动机的状态图 DFA (它是 M 的等价确定型自动机, 称作 DM) 中的节点是 M 中节点的集合。DM 的开始节点是 M 开始节点的 λ 闭包。算法的核心是第 2.1.1 步, 它产生确定型自动机的节点。如果 X 是 DM 中的一个节点, 那么集合 Y 构造如下: 它包含从集合 X 中的任意状态通过处理符号 a 能够进入的所有状态。此关系在 DM 的状态图中表示为从 X 到 Y 的一条标记为 a 的弧。通过为字母表之中的所有的符号构造一条从 X 出发的弧, 这样 X 也是确定的了。在 2.1.1 步中产生的新的节点被加入到集合 Q' 中, 处理会继续进行, 直到 Q' 中的所有节点都是确定的。 [171]

算法 5.6.3 构造与 NFA- λ M 等价的 DFA DM

输入: NFA- λ $M = (Q, \Sigma, \delta, q_0, F)$

M 的输入转换函数 t

1. 将 Q' 初始化为 $\lambda\text{-closure}(q_0)$

2. repeat

2.1. if 存在节点 $X \in Q'$ 并且没有标记为 a 的弧从节点 X 出发, $a \in \Sigma$, then

2.1.1. 置 $Y = \bigcup_{q_i \in X} t(q_i, a)$

2.1.2. if $Y \notin Q'$, then 置 $Q' := Q' \cup \{Y\}$

2.1.3. 增加一条从 X 到 Y 的标记为 a 的弧

else done; = true

until done

3. DM 的接收状态的集合是 $F' = \{X \in Q' \mid X \text{ 包含一个元素 } q_i \in F\}$

使用例 5.6.1 中的 NFA- λ 来演示等价的 DFA 节点的构造。DM 的开始节点是包含 M 的开始节点的一个集合。从状态 q_0 处理一个 a 的转换可以停止在状态 q_0, q_1 或 q_2 。因此, 为 DFA 构造节点 $\{q_0, q_1, q_2\}$, 并用一条标记为 a 的弧把它连接到 q_0 上。DM 中从 q_0 到 $\{q_0, q_1, q_2\}$ 的路径表示, 在 M 中

的状态 q_0 有三种可能的方法处理符号 a 。

因为 DM 是确定的, 所以节点 $\{q_0\}$ 必须有标记为 b 和 c 的弧离开。增加从节点 q_0 到 \emptyset 标记为 b 和 c 的弧, 这表明在 NFA- λ M 在状态 q_0 扫描到这些字符的时候没有指定的动作。

节点 $\{q_0\}$ 具有确定的形式: 对字母表中的每一个元素存在, 惟一的标记为这些元素的离开此节点的弧。图 5-4 (a) 展示构造的这个阶段。这个阶段创造了两个额外的节点 $\{q_0, q_1, q_2\}$ 和 \emptyset 。这两个节点都必须是确定的。

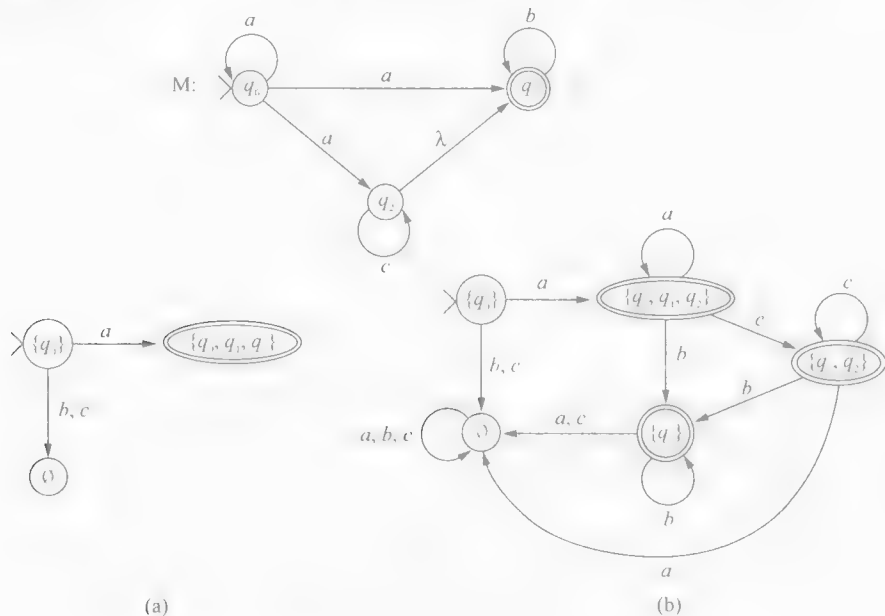


图 5-4 等价确定型自动机的构造

从节点 $\{q_0, q_1, q_2\}$ 离开的弧停止在这样的一个节点, 该节点包含所有的从状态 q_0, q_1 或 q_2 经过处理输入符号能够达到的状态。输入转换函数 $t(q_i, a)$ 指定了从状态 q_i 经过处理输入 a 能够到达的状态。从 $\{q_0, q_1, q_2\}$ 出发的标记为 a 的弧, 终止在由 $t(q_0, a)$ 、 $t(q_1, a)$ 和 $t(q_2, a)$ 的并组成的集合。这个并的结果仍然是 $\{q_0, q_1, q_2\}$ 。在这个转换的状态图中添加了一条从 $\{q_0, q_1, q_2\}$ 指向自身的弧。

空集表示 DM 的一个错误状态。在状态 \emptyset 读入 a 进入空集, 当且仅当对 a 和任意的 $q_i \in \emptyset$ 不存在转换。一旦进入 \emptyset , 计算将处理剩下的输入, 并拒绝接收串。这在状态图中表示为从 \emptyset 指向自身的标记为每一个字母表中符号的弧。

图 5-4 (b) 给出了和 M 等价的完整的确定型自动机。非确定机对输入串 aaa 的计算可以终止于状态 q_0, q_1 和 q_2 。终止在状态 q_1 的路径展示了串 aaa 的接收。DM 中处理 aaa 终止在状态 $\{q_0, q_1, q_2\}$, 这个状态在 DM 中是接收状态, 因为它包含 M 的接收状态 q_1 。

构造确定型自动机的状态图的算法包含重复地增加弧, 从而使得图中的节点是确定性的。在构造弧的时候, 可能构造新的节点并把它加入到状态图之中。当所有的节点都是确定的时候, 算法终止。因为每个节点是 Q 的一个子集, 因此最多可以构造 $\text{card}(\mathcal{P}(Q))$ 个节点。因为 $\text{card}(\mathcal{P}(Q)) \times \text{card}(\Sigma)$ 是迭代次数的上限, 所以算法 5.6.3 总是会终止的。定理 5.6.4 将说明 M 和 DM 是等价的。

定理 5.6.4 设 $w \in \Sigma^*$ 并且 $Q_w = \{q_n, q_{n+1}, \dots, q_n\}$ 是 M 处理串 w 完成时进入的状态的集合。那么, 在 DM 中处理 w 会终止在状态 Q_w 。

证明: 证明将对串 w 的长度进行归纳。M 处理完空串后终止在节点 $\lambda\text{-closure}(q_0)$ 。这个集合是 DM 的开始状态。

假设对于所有的长度为 n 的串, 该性质都成立。设 $w = ua$ 是一个长度为 $n+1$ 的串。设 $Q_u = \{q_u, q_{u+1}, \dots, q_u\}$ 是 M 通过处理输入串 u 得到的路径的最终状态。由归纳假设, DM 处理 u 终止在 Q_u 。M 处理

ua 的计算终止于在 Q_u 中处理 a 能够进入的这些状态上。集合 Q_w 可以通过如下的输入转换函数来定义：

$$Q_w = \bigcup_{i=1}^k \delta(q_u, a).$$

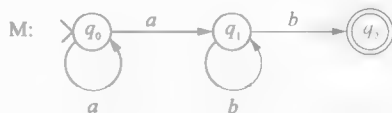
因为 Q_w 是 DM 在状态 Q_u 下经过处理输入 a 进入的状态，证毕。 ■

非确定型自动机接收一个串，这依赖于是否存在一个处理完整串并终止在接收状态的计算。节点 Q_u 包含由 M 产生的接收 w 的所有路径的终止状态。如果 w 被 M 接收，那么 Q_u 包含 M 的一个接收状态。这个接收节点使 Q_u 成为了 DM 的一个接收状态，根据前面的定理， w 被 DM 接收。

另一方面，如果 w 被 DM 接收，那么 Q_u 包含 M 的一个接收状态。 Q_u 的构造保证了 M 中存在一个处理完 w 并且终止在接收状态的计算。这些观察为推论 5.6.5 提供了证明。

推论 5.6.5 有限自动机 M 和 DM 是等价的。

例 5.6.2 NFA

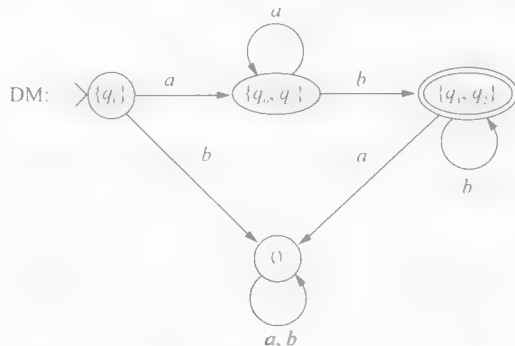


接收语言 a^+b^+ ，下表中构造了一个等价的 DFA：

[174]

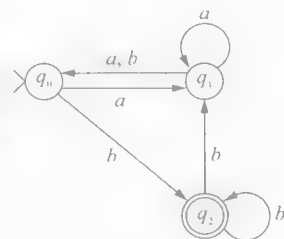
状态	符号	NFA 转换	下一个状态
$\{q_0\}$	a	$\delta(q_0, a) = \{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0\}$	b	$\delta(q_0, b) = \emptyset$	\emptyset
$\{q_0, q_1\}$	a	$\delta(q_0, a) = \{q_0, q_1\}$ $\delta(q_1, a) = \emptyset$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	b	$\delta(q_0, b) = \emptyset$ $\delta(q_1, b) = \{q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	a	$\delta(q_1, a) = \emptyset$ $\delta(q_2, a) = \emptyset$	\emptyset
$\{q_1, q_2\}$	b	$\delta(q_1, b) = \{q_1, q_2\}$ $\delta(q_2, b) = \emptyset$	$\{q_1, q_2\}$

因为 M 是 NFA，所以 M 的转换函数 δ 作为输入转换函数的、等价的 DFA 的开始状态是 $\{q_0\}$ 。最后得到的 DFA 为：

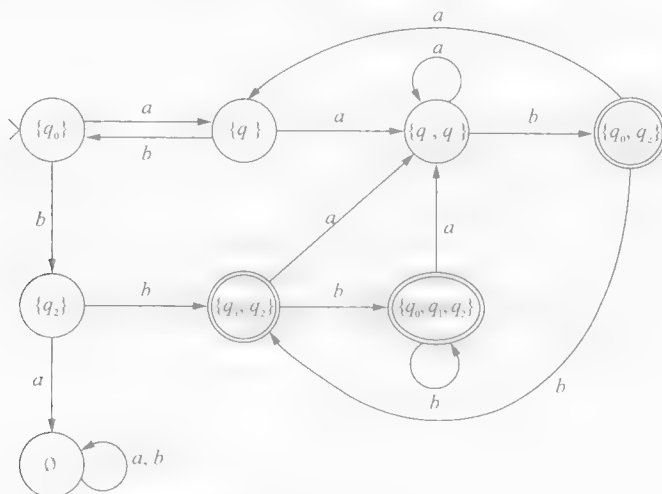


例 5.6.3 正如在前面的例子中见到的，利用算法 5.6.3 构造的 DFA 的状态是原来的非确定型自动机的状态集合。如果非确定型自动机有 n 个状态，那么 DFA 可能有 2^n 个状态。NFA 的转换表明，状态数目的理论上界是可以获得的。

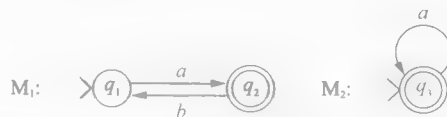
DM 的开始状态是 $\{q_0\}$ ，因为 M 不含有 λ -转换。



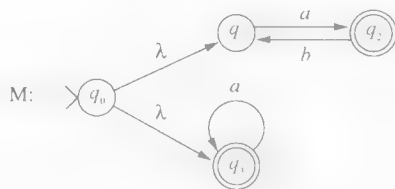
[175]



例 5.6.4 自动机 M_1 和 M_2 分别接收语言 $a(ba)^*$ 和 a^* 。



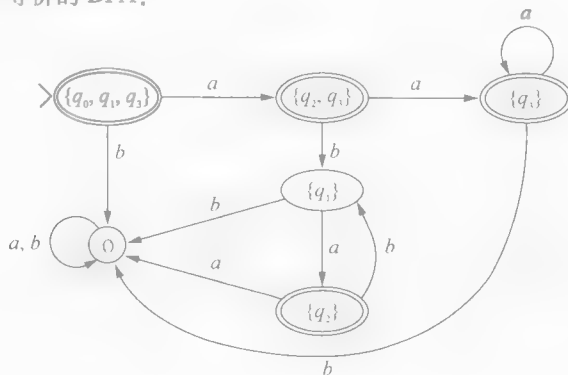
使用 λ -弧将一个新状态连接到原始的自动机的开始状态上，这样就产生了一个新的 NFA- λ M ，它接收 $a(ba)^* \cup a^*$ 。



M 的输入转换函数如下所示：

t	a	b
q_0	$\{q_2, q_3\}$	\emptyset
q_1	$\{q_2\}$	\emptyset
q_2	\emptyset	$\{q_1\}$
q_3	$\{q_3\}$	\emptyset

使用算法 5.6.3 可以获得等价的 DFA：



算法 5.6.3 完成了下图中所描述的各种有限自动机之间的关系。



其中, 箭头表示包含关系; 每一个 DFA 能够被再次形式化为 NFA 以及 NFA- λ 。从 NFA- λ 到 DFA 的双箭头表明存在一个与 NFA- λ 等价的确定型自动机。

[177]

5.7 DFA 的最小化

前面的小节说明了 DFA、NFA 以及 NFA- λ 接收的语言是一样的。非确定型和 λ -转换使得设计自动机接收复杂的语言较为容易。采用算法 5.6.3 可以将非确定型的自动机转换为等价的确定型自动机。然而得到的 DFA 可能不是接收语言的最小的 DFA。本节将展示一个化简算法, 它能够从一个接收语言 L 的 DFA 获得一个接收该语言的最小的 DFA。为了完成化简, 下面先介绍 DFA 中等价状态的记号。

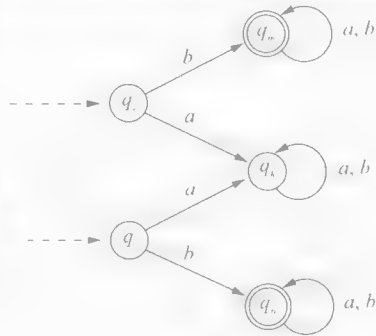
定义 5.7.1 设 $M = (Q, \Sigma, \delta, q_0, F)$ 是 DFA。如果, 对每一个 $u \in \Sigma^*$, $\hat{\delta}(q_i, u) \in F$ 当且仅当 $\hat{\delta}(q_j, u) \in F$, 那么, 状态 q_i 和 q_j 等价。

两个等价的状态称为不可区分的 (indistinguishable)。状态的不可区分性定义的 Q 上的二元关系是一个等价关系; 即, 该关系是自反的、对称的和传递的。两个不等价的状态称为是可区分的。状态 q_i 和 q_j 是可区分的, 如果存在串 u 使得 $\hat{\delta}(q_i, u) \in F$ 并且 $\hat{\delta}(q_j, u) \notin F$, 或者 $\hat{\delta}(q_i, u) \notin F$ 并且 $\hat{\delta}(q_j, u) \in F$ 。

如右图所示的状态和转换展示了等价定义背后的动机。

进入 q_i 和 q_j 的不带标记的虚线表明进入该状态是无关紧要的; 等价只依赖于状态上的计算。状态 q_i 、 q_j 是等价的, 因为它们在以任何 b 开始的输入串上的计算都进入一个接收状态, 而任何以 a 开始的串的计算都终止在一个非接收状态。因为状态 q_m 和 q_n 是等价的, 所以状态 q_i 和 q_j 也是等价的。

转换背后的直觉告诉我们, 可以合并等价的状态。将此应用到前面的例子可得到如下图所示的状态和转换。

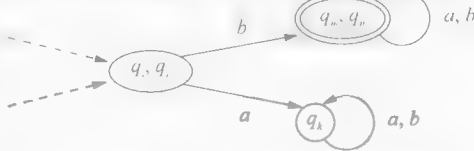


[178]

通过合并来减少 DFA 的状态, 需要设计一个识别等价状态的过程。在完成这项工作的算法中, 每一对状态 q_i 和 q_j ($i < j$) 具有一对相关联的值 $D[i, j]$ 和 $S[i, j]$ 。当确定两个状态是可区别的时候, $D[i, j]$ 被设置为 1。 $S[m, n]$ 是索引的集合。当状态 q_i 和 q_j 的区分能够由状态 q_m 和 q_n 的区分得出时, 索引 $[i, j]$ 在集合 $S[m, n]$ 中。

算法开始时, 如果 q_i 和 q_j 中一个是接收状态而另

一个不是接收状态, 则将每一个状态对 q_i 和 q_j 标记为可区分的。接下来, 算法系统地检查每一个没有标记的状态对。当两个状态标明是可以区分的时, 递归的调用过程 $DIST$ 将 $D[i, j]$ 设置为 1。调用 $DIST(i, j)$ 不只是标记 q_i 和 q_j 是可区分的, 同时还通过调用 $DIST[m, n]$ 来标记 $S[i, j]$ 中的每一个状态对 q_m 和 q_n 为可区分的。



算法 5.7.2 确定 DFA 的等价状态

输入: DFA $M(Q, \Sigma, \delta, q_0, F)$

1. (初始化)

for 每一状态对 q_i 和 q_j , $i < j$, do

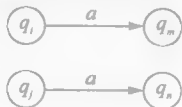
1.1. $D[i, j] := 0$

```

1. 2.  $S[i, j] := \emptyset$ 
   end for
2. for 每一对  $i, j, i < j$ , if  $q_i$  和  $q_j$  中的一个接收状态, 另一个不是接收状态 then 置  $D[i, j] := 0$ 
3. for 每一对  $i, j, i < j$ , 并且  $D[i, j] := 0$ , do
    3. 1. if 存在  $a \in \Sigma, \delta(q_i, a) = q_m, \delta(q_j, a) = q_n$ , 并且  $D[m, n] = 1$  或者  $D[n, m] = 1$ , then
         $DIST(i, j)$ 
    3. 2. else 对每一个  $a \in \Sigma$ , 执行: 置  $\delta(q_i, a) = q_m$  并且  $\delta(q_j, a) = q_n$ 
        if  $m < n$  并且  $[i, j] \neq [m, n]$ , then 将  $[i, j]$  加入到  $S[m, n]$  中,
        else if  $m > n$  并且  $[i, j] \neq [n, m]$ , then 将  $[i, j]$  加入到  $S[n, m]$  中
    end for
     $DIST(i, j)$ ;
begin
     $D[i, j] := 1$ 
    for 所有的  $[m, n] \in S[i, j], DIST(m, n)$ 
end

```

下图之间的关系展示了可区分状态标识背后的动机:



当在第3步之中检查 q_i 和 q_j 的时候, 如果 q_m 和 q_n 已经标识为可区分, 那么 $D[i, j]$ 被设置为 1 来表明状态 q_i 和 q_j 是可区分的。如果当检查 q_i 和 q_j 的时候, q_m 和 q_n 的状态未知, 那么之后当决定的 q_m 和 q_n 是可以区分的时, 同样可以确定 q_i 和 q_j 是可区分的。数组 S 是用来记录信息 $[i, j] \in S[n, m]$, 这表明 q_m 和 q_n 的可区分性足够确定状态 q_i 和 q_j 是否是可区分的。这些思想的形式化见定理 5.7.3 的证明。

定理 5.7.3 状态 q_i 和 q_j 是可区分的, 当且仅当在算法 5.7.2 终止的时候 $D[i, j] = 1$ 。

证明: 首先, 我们将展示每一对满足 $D[i, j] = 1$ 的状态 q_i 和 q_j 都是可以区分的。如果 $D[i, j]$ 在第2步之中被设置为 1, 那么状态 q_i 和 q_j 可以被空串区分。在算法已经将 q_m 和 q_n 确定为是可以区分的情况下, 如果对于某些输入 $a, \delta(q_i, a) = q_m$ 和 $\delta(q_j, a) = q_n$ 都成立, 第3.1步才标记 q_i 和 q_j 是可区分的。设 u 是能够区分 q_m 和 q_n 的串, 那么 au 将能够区分状态 q_i 和 q_j 。

为了完成证明, 我们有必要展示每一个可区分状态对是如何被指派。证明通过对能够区分可区分状态对的最短串的长度进行归纳。归纳的基础包含所有的能够被长度为 0 的串区分的状态对 q_i 和 q_i 。即, 计算 $\hat{\delta}(q_i, \lambda) = q_i$ 和 $\hat{\delta}(q_i, \lambda) = q_i$ 将把 q_i 和 q_i 区分开。在这种情况下, q_i 和 q_i 中只有一个状态是接收状态, $D[i, j]$ 在第2步之中被置为 1。

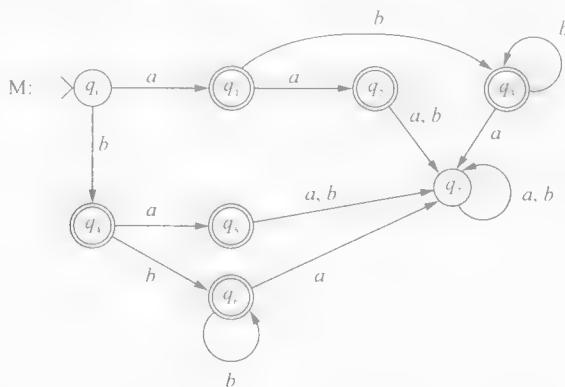
现在, 假设算法已经标记出了所有的能够被长度不大于 k 的串区分开的状态对。设 q_i 和 q_j 的最短区分串 u 的长度为 $k+1$ 。那么, u 可以写成 av , 输入为 u 的计算具有下面的形式: $\hat{\delta}(q_i, u) = \hat{\delta}(q_i, av) = \hat{\delta}(q_m, v) = q_i$ 和 $\hat{\delta}(q_j, u) = \hat{\delta}(q_j, av) = \hat{\delta}(q_n, v) = q_i$ 。因为前面的计算已经区分了 q_i 和 q_j , 所以 q_i 和 q_j 中只有一个接收状态。很明显, 同样的计算展示了 q_m 和 q_n 被一个长度为 k 的串区分。根据归纳, 我们知道算法将置 $D[m, n]$ 为 1。

如果在第3步检查状态 q_i 和 q_j 之前, $D[m, n]$ 已经被置为 1, 那么会通过调用 $DIST(i, j)$ 将 $D[i, j]$ 置为 1。如果在 3.1 步的循环之中检查状态 q_i 和 q_j , 并且此时 $D[m, n] \neq 1$, 那么将会把 $[i, j]$ 添加到集合 $S[m, n]$ 之中。根据归纳假设, $D[m, n]$ 将最终被置为 1。因为 $[i, j]$ 在集合 $S[m, n]$ 之中, 通过递归调用 $DIST(m, n)$ 将置 $D[i, j]$ 为 1。

从原始的 DFA $M = (Q, \Sigma, \delta, q_0, F)$ 和不可区分关系可以构造出一个新的 DFA M' 。 M' 的状态是 M 之中的不可区分状态构成的等价类。初始状态是 $[q_0]$, 终止状态是 $[q_i]$, 其中 $q_i \in F$ 。 M' 的转换函数

δ' 定义为 $\delta'([q_i], a) = [\delta'(q_i, a)]$ 。在练习 44 中展示了良好定义的 δ' 。 $L(M')$ 包括所有的具有如下计算形式的串: $\hat{\delta}([q_0], u) = [\hat{\delta}(q_i, \lambda)]$, 其中 $q_i \in F$ 。它们是正好被 M 接收的串。如果 M' 之中具有从 $[q_0]$ 通过计算不可达的状态, 那么会删除这些状态和与它们相关的所有弧。

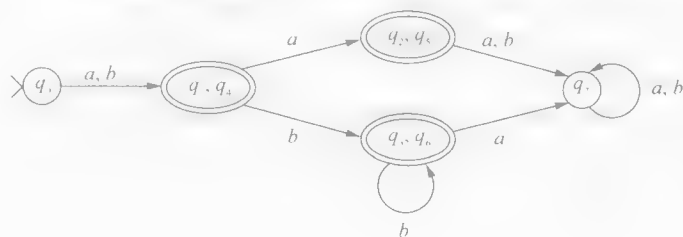
例 5.7.1 下面展示了 DFA M 的最小化过程, M 接收的语言为 $(a \cup b)(a \cup b)^*$ 。



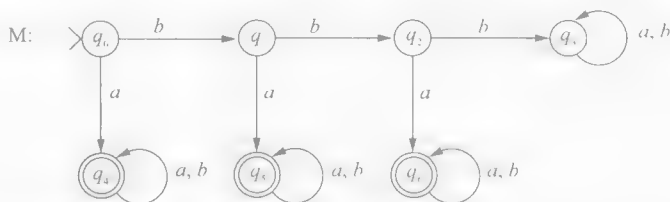
在第 2 步中, $D[0,1]$ 、 $D[0,2]$ 、 $D[0,3]$ 、 $D[0,4]$ 、 $D[0,5]$ 、 $D[0,6]$ 、 $D[1,7]$ 、 $D[2,7]$ 、 $D[3,7]$ 、 $D[4,7]$ 、 $D[5,7]$ 和 $D[6,7]$ 被置为 1。每一个在第 2 步中没有标记的索引将在第 3 步中得到检查。下面的表格展示了每一个索引所采取的动作:

索引	动作	原因
$[0,7]$	$D[0,7] = 1$	被 a 区分
$[1,2]$	$D[1,2] = 1$	被 a 区分
$[1,3]$	$D[1,3] = 1$	被 a 区分
$[1,4]$	$S[2,5] = \{[1,4]\}$ $S[3,6] = \{[1,4]\}$	
$[1,5]$	$D[1,5] = 1$	被 a 区分
$[1,6]$	$D[1,6] = 1$	被 a 区分
$[2,3]$	$D[2,3] = 1$	被 b 区分
$[2,4]$	$D[2,4] = 1$	被 a 区分
$[2,5]$		无动作, 因为 $\delta(q_2, x) = \delta(q_5, x), x \in \Sigma$
$[2,6]$	$D[2,6] = 1$	被 b 区分
$[3,4]$	$D[3,4] = 1$	被 a 区分
$[3,5]$	$D[3,5] = 1$	被 b 区分
$[3,6]$		
$[4,5]$	$D[4,5] = 1$	被 a 区分
$[4,6]$	$D[4,6] = 1$	被 a 区分
$[5,6]$	$D[5,6] = 1$	被 b 区分

在对每一对索引检查之后, 会剩下 $[1,4]$ 、 $[2,5]$ 和 $[3,6]$, 它们是等价状态对。合并这些状态得到一个接收 $(a \cup b)(a \cup b)^*$ 的最小状态的 DFA M' 。



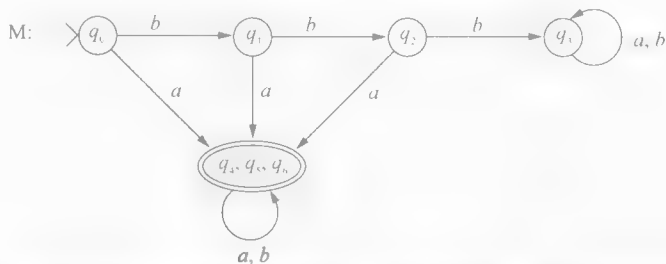
例 5.7.2 最小化 DFA M 展示了通过调用 $DIST$ 递归的标识状态的过程。 M 的语言是 $a(a \cup b)^* \cup ba(a \cup b)^* \cup bba(a \cup b)^*$ 。



由于接收状态和非接收状态可区分，所以将 $D[0,4]$ 、 $D[0,5]$ 、 $D[0,6]$ 、 $D[1,4]$ 、 $D[1,5]$ 、 $D[1,6]$ 、 $D[2,4]$ 、 $D[2,5]$ 、 $D[2,6]$ 、 $D[3,4]$ 、 $D[3,5]$ 和 $D[3,6]$ 都置为 1。跟踪算法将会产生下面的表格：

索引	动作	原因
[0,1]	$S[4,5] = \{[0,1]\}$ $S[1,2] = \{[0,1]\}$	
[0,2]	$S[4,6] = \{[0,2]\}$ $S[1,3] = \{[0,2]\}$	
[0,3]	$D[0,3] = 1$	被 a 区分
[1,2]	$S[5,6] = \{[1,2]\}$ $S[2,3] = \{[1,2]\}$	
[1,3]	$D[1,3] = 1$ $D[0,2] = 1$	被 a 区分 调用 $DIST(1,3)$
[2,3]	$D[2,3] = 1$ $D[1,2] = 1$ $D[0,1] = 1$	被 a 区分 调用 $DIST(1,2)$ 调用 $DIST(0,1)$
[4,5]		
[4,6]		
[5,6]		

合并等价状态 q_4 、 q_5 和 q_6 会产生



最小化算法完成了构造最优 DFA 的算法的序列。非确定性和 λ -转换为设计匹配复杂模式或者接收复杂语言的自动机提供了工具。然后，采用算法 5.6.3 将非限定机转化为 DFA，但这个 DFA 可能不是最小的。算法 5.7.2 完成了产生最小化 DFA 的过程。

到此，我们已经展示了化简 DFA 的算法，但是这个算法并没有证明得到的就是最小的 DFA。在 6.7 节之中，我们将给出 Myhill-Nerode 定理，它以字符串等价类的形式刻画了一个有限自动机接收的语言。这个特征将用来证明算法 5.7.2 产生的自动机 M' 是接收 L 的惟一的最小状态 DFA。

5.8 练习

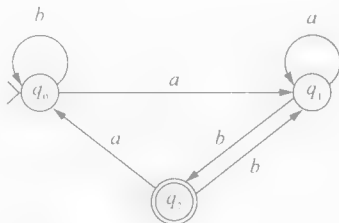
1. 设 M 是确定型有限状态自动机, 它的定义如下:

$Q = \{q_0, q_1, q_2\}$	δ	a	b
$\Sigma = \{a, b\}$	q_0	q_0	q_1
$F = \{q_2\}$	q_1	q_2	q_1
	q_2	q_2	q_0

- 给出 M 的状态图。
 - 跟踪 M 处理串 $abaa$ 、 $bbbabb$ 、 $bababa$ 和 $bbbaa$ 的计算过程。
 - (b) 中的哪些串能够被 M 接收?
 - 给出 $L(M)$ 的正则表达式。
2. 设 M 是确定的有限状态自动机

$Q = \{q_0, q_1, q_2\}$	δ	a	b
$\Sigma = \{a, b\}$	q_0	q_1	q_0
$F = \{q_0\}$	q_1	q_1	q_2
	q_2	q_1	q_0

- 给出 M 的状态图。
 - 跟踪 M 处理串 $babaab$ 的过程。
 - 给出 $L(M)$ 的正则表达式。
 - 如果 q_0 和 q_1 是接收状态, 那么给出 M 接收语言的正则表达式。
3. DFA M 的状态图如下:

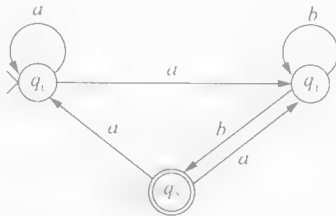


- 构造 M 的转换表。
 - $baba$ 、 $baab$ 、 $abab$ 和 $abaaab$ 中哪些能够被 M 接收?
 - 给出 $L(M)$ 的正则表达式。
4. 扩展转换函数定义 (定义 5.2.4) 中的递归步骤可以用 $\hat{\delta}'(q_i, au) = \hat{\delta}'(\delta(q_i, a), u)$ 替换, 对所有的 $u \in \Sigma^*$ 、 $a \in \Sigma$ 和 $q_i \in Q$ 。证明: $\hat{\delta} = \hat{\delta}'$ 。
- 练习 5 到练习 21, 构造一个接收所描述的语言的 DFA。
- $\{a, b, c\}$ 上的字符串, 满足所有的 a 在 b 之前, 所有的 b 在 c 之前, 也可能没有 a 、 b 或 c 。
 - $\{a, b\}$ 上的字符串的集合, 其中子串 aa 至少出现两次。
 - $\{a, b\}$ 上的不以 aaa 开头的字符串的集合。
 - $\{a, b\}$ 上的不包含 aaa 子串的字符串的集合。
 - $\{a, b, c\}$ 上的以 a 开头, 包含 2 个 b 并且以 cc 结尾的字符串的集合。
 - $\{a, b, c\}$ 上的字符串的集合, 满足每一个 b 之后至少有一个 c 。
 - $\{a, b\}$ 上的字符串的集合, 满足 a 的个数能够被 3 整除。
 - $\{a, b\}$ 上的字符串的集合, 满足每一个 a 直接在一个 b 之前或者 b 之后, 如: $baab$ 、 aba 和 b 。
 - $\{a, b\}$ 上的包含 bb 子串的长度为奇数的串的集合。
 - $\{a, b\}$ 上的字符串的集合, 满足长度是奇数或者以 aaa 结尾。

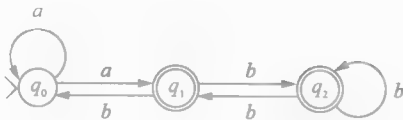
15. $\{a, b, c\}$ 上的只包含一个 a 并且长度是偶数的字符串的集合。
16. $\{a, b\}$ 上的字符串的集合, 满足子串 aa 出现的次数为奇数。注意: aaa 中 aa 出现了两次。
17. $\{a, b\}$ 上的包含有偶数个 ba 子串的字符串的集合。
18. $\{1, 2, 3\}$ 上的串的集合, 满足构成串的所有数字的和能够被 6 整除。
19. $\{a, b, c\}$ 上的字符串的集合, 满足 a 的数目加上 b 的数目再加上 c 的数目的两倍能够被 6 整除。
20. $\{a, b\}$ 上的字符串的集合, 满足长度为 4 的子串之中至少含有一个 b 。注意: 每一个长度小于 4 的串在该语言之中。
21. $\{a, b\}$ 上的字符串的集合, 满足每一个长度为 4 的子串之中都有且只有一个 b 。
22. 对于下面的每一个语言, 给出接收这些语言的正 DFA 的状态图。

- a) $(ab)^*ba$
- b) $(ab)^*(ba)^*$
- c) $aa(a \cup b)^*bb$
- d) $((aa)^*bb)^*$
- e) $(ab^*a)^*$

23. 设 M 是非确定型有限状态自动机



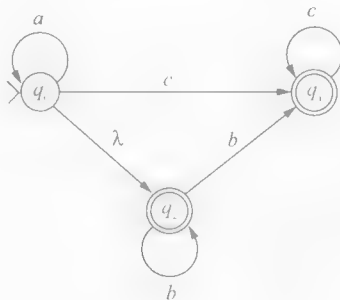
- a) 构造 M 的转换表。
- b) 跟踪 M 中对 $aaabb$ 的所有计算过程。
- c) $aaabb$ 在 $L(M)$ 之中吗?
- d) 给出 $L(M)$ 的正则表达式。
24. 设 M 是非确定有限自动机



- a) 构造 M 的转换表。
- b) 跟踪 M 计算 $aabb$ 的过程。
- c) $aabb$ 在 $L(M)$ 之中吗?
- d) 给出 $L(M)$ 的正则表达式。
- e) 构造一个接收 $L(M)$ 的 DFA。
- f) 如果 q_0 和 q_1 是接收状态, 给出 M 接收语言的正则表达式。
25. 对于下面的每一个语言, 给出接收这些语言的正 NFA 的状态图。
- a) $(a \cup ab \cup aab)^*$
- b) $(ab)^* \cup a^*$
- c) $(abc)^*a^*$
- d) $(ba \cup bb)^* \cup (ab \cup aa)^*$
- e) $(ab^*a)^*$
26. 给出一个 NFA- λ 的扩展转换函数 $\hat{\delta}$ 的递归定义。 $\hat{\delta}(q_i, w)$ 是从节点 q_i 开始, 并能够完全处理完串 w 的计算所能够到达的状态的集合。

练习 27 到练习 34, 给出接收给定语言的 NFA 的状态图。记住 NFA 可以是确定的, 但是在任何适当的地方你应该使用非确定性。

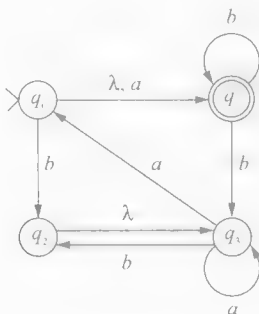
27. $\{a, b\}$ 上的包含 aa 或者 bb 子串的字符串的集合。
28. $\{a, b\}$ 上的同时包含或者同时不包含子串 aa 和 bb 的字符串的集合。
- *29. $\{a, b\}$ 上的字符串的集合, 满足从第三个到最后一个符号都是 b 。
30. $\{a, b\}$ 上的字符串的集合, 其中第三个和倒数第三个字符都是 b 例如, $aababaa$ 、 $abbbbbb$ 和 $abba$ 都在该语言之中。
31. $\{a, b\}$ 上的字符串的集合, 满足每一个 a 后面都跟着 b 或 ab 。
32. $\{a, b\}$ 上的字符串的集合, 具有一个长度为 4 的子串, 该子串从开始到结束都是同一个字符。
33. $\{a, b\}$ 上的字符串的集合, 满足包含子串 aaa 和 bbb 。
34. $\{a, b, c\}$ 上的字符串的集合, 这些串具有一个长度为 3 的子串, 在该子串之中 a 、 b 和 c 都恰好出现一次。
35. 构造一个 DFA, 该 DFA 接收 $\{a, b\}$ 上的以 $abba$ 结束的串。并给出接收同一语言的具有 6 条弧的 NFA 状态图。
36. 设 M 是 NFA- λ



- a) 计算 $\lambda\text{-closure}(q_i)$, $i=0, 1, 2$ 。
- b) 给出 M 的输入转换函数 t 。
- c) 利用算法 5.6.3 构造一个与 M 等价的 DFA 状态图。
- d) 给出 $L(M)$ 的正则表达式。

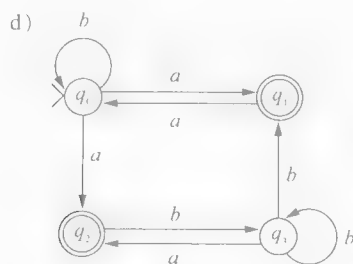
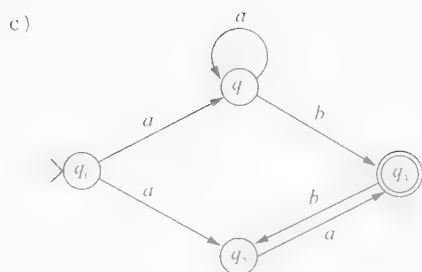
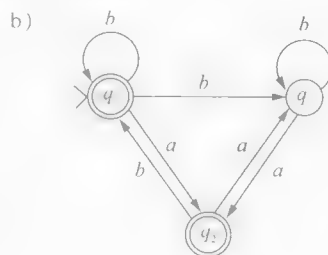
[187]

37. 设 M 是 NFA- λ ,



- a) 计算 $\lambda\text{-closure}(q_i)$, $i=0, 1, 2, 3$ 。
- b) 给出 M 的输入转换函数 t 。
- c) 利用算法 5.6.3 构造一个与 M 等价的 DFA 的状态图。
- d) 给出 $L(M)$ 的正则表达式。
38. 利用算法 5.6.3 构造一个与例 5.5.3 中的 NFA 等价的 DFA 状态图。
39. 利用算法 5.6.3 构造一个与练习 17 中的 NFA 等价的 DFA 状态图。

40. 对下面的每一个 NFA，利用算法 5.6.3 构造一个等价的 DFA 状态图。



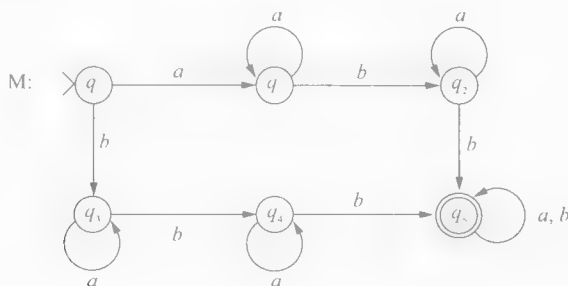
41. 构造一个接收 $(ab)^*$ 的 NFA M_1 和一个接收 $(ba)^*$ 的 NFA M_2 ，利用 λ -转换获得一个接收 $(ab)^*(ba)^*$ 的自动机 M ，给出 M 的输入转换函数，利用算法 5.6.3 构造一个接收 $L(M)$ 的 DFA 状态图。
42. 构造一个接收 $(aba)^*$ 的 NFA M_1 和一个接收 $(ba)^*$ 的 NFA M_2 ，利用 λ -转换获得一个接收 $(aba)^* \cup (ab)^*$ 的自动机 M ，给出 M 的输入转换函数，利用算法 5.6.3 构造一个接收 $L(M)$ 的 DFA 状态图。
43. 设状态 q_i 和 q_j 是 DFA M 的等价状态（如定义 5.7.1），并且 $\hat{\delta}(q_i, u) = q_m$ 和 $\hat{\delta}(q_j, u) = q_n$ ，对每一个 $u \in \Sigma^*$ 都成立。证明： q_m 和 q_n 是等价的。

44. 说明在合并等价状态的过程之中获得的转换函数 δ' 是良定义的。即，说明如果状态 q_i 和 q_j 是满足 $[q_i] = [q_j]$ 的状态，那么 $\delta'([q_i], a) = \delta'([q_j], a)$ ，对每一个 $a \in \Sigma$ 都成立。

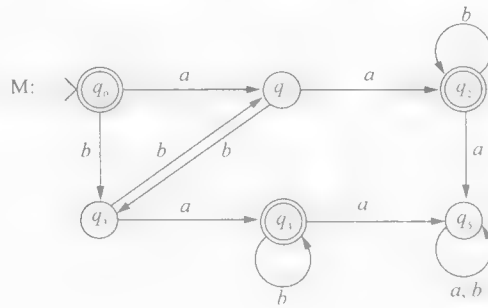
45. 对每一个 DFA：

- 跟踪算法 5.7.2 的动作，确定 M 的等价状态，并给出算法中计算的 $D[i, j]$ 和 $S[i, j]$ 的值。
- 给出状态的等价类。
- 给出接收 $L(M)$ 的最小状态 DFA 状态图。

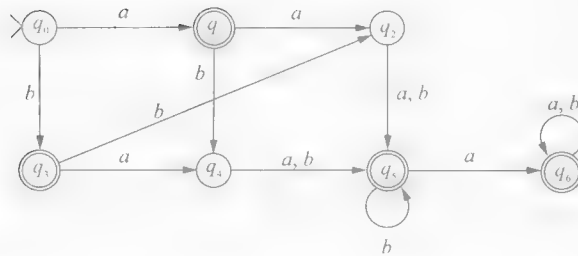
a)



b)



c)



参考文献注释

Mealy [1955] 和 Moore [1956] 研究了有限状态计算结果的另外一个表示。Mealy 机之中的转换伴随着输出的产生。一个双向自动机允许带头朝着两个方向移动。接收相同语言的双向和单向自动机的证明可以在 Rabin 和 Scott [1959] 中找到，在 Rabin 和 Scott [1959] 中介绍了非确定型自动机。Nerode [1958] 中给出了最小化 DFA 状态数的算法。Hopcroft [1971] 中的算法提高了最小化技术的效率。

有限自动机的理论和应用在 Minsky [1967]、Salomaa [1973]、Denning, Denning 和 Bravel [1983] 之中得到了巨大的发展。

第6章 正则语言的性质

文法作为语言的产生器而被引入，有限自动机作为语言的接收器，而正则表达式则作为模式描述符。本章将介绍这三种语言定义方法之间的关系，同时将探究有限自动机作为语言接收器的限制。

6.1 有限状态机接收正则语言

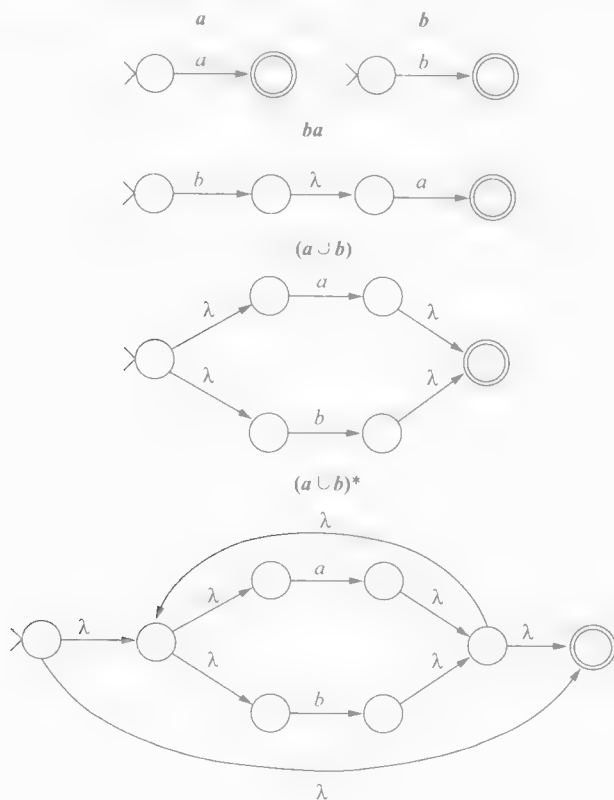
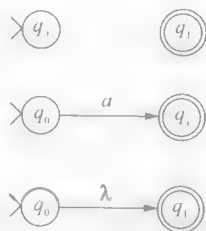
本节将介绍如何构造一个接收正则语言的 $\text{NFA}-\lambda$ 。正则集合递归地从 \emptyset 、 λ 和一个单元素集合中构造出来，这个过程将会应用并、连接和 Kleene 星操作（定义 2.3.2）。构造接受正则集合的 $\text{NFA}-\lambda$ 可以通过下面的几个递归产生步骤获得。我们将采用状态图而不是集合来构造。

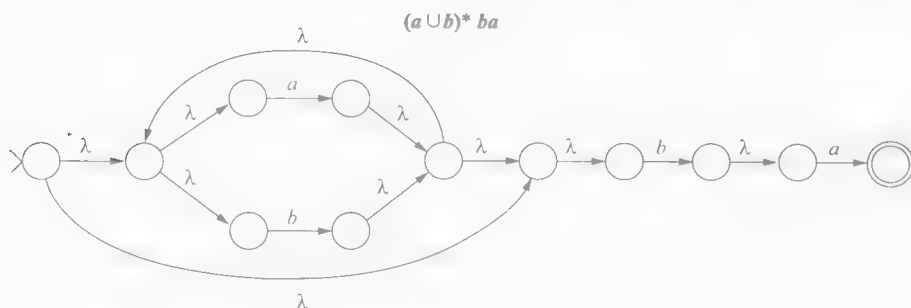
接受 \emptyset 、 λ 和单元素集合 $|a|$ 的自动机的状态图如下：

注意每一个自动机满足引理 5.5.2 描述的约束，即，自动机包括惟一的接收状态，并且没有弧进入开始状态或者离开接收状态。

正如定理 5.5.3 所指出的，可以利用 λ -转换来连接这种形式的自动机，从而产生新的接收更加复杂的语言的自动机。重复地利用这一技术，相应的机器操作将能够模仿从基本元素构造正则表达式。这个过程将通过下面的例子予以阐明。

例 6.1.1 通过递归定义的正则表达式的方法构造一个接收 $(a \cup b)^* ba$ 的 $\text{NFA}-\lambda$ 。中间机器接收的语言标识在状态图的上面。





□

6.2 表达式图

前一节中的构造表明了每一个正则语言都可以用一个有限自动机来重新组织。下面我们将展示每一个有限自动机接收的语言都是正则的，并为自动机接收的语言构造一个正则表达式。为了完成这个任务，我们需要扩展状态图的概念。

定义 6.2.1 一个表达式图 (expression graph) 是一个带标记的有向图，每条弧上都标有正则表达式。表达式图和状态图一样，包含一个初始节点和一组接收节点。

字母表为 Σ 的有限自动机的状态图是表达式图的一个特例；标记包含 λ 和字母表中的元素。表达式图中的路径产生正则表达式。表达式图的语言是从开始节点到接收节点的正则表达式的并。例如，表达式图



接收的语言分别为： $(ab)^*$ 、 $(b^+a)^*(a \cup b)(ba)^*$ 和 $(ba)^*b^*(bb \cup (a^+(ba)^*b^+))^*$ 。

由于图非常简单，因此前面的例子接收的语言的表达式是显然的。人们开发了一个过程用来将任意的表达式图化简为最多只包含两个节点的表达式图。化简是通过反复地从图中删除节点，同时又保证图接收的语言不变。

有限自动机的状态图可能含有任意数目的接收状态。每一个状态展示了接收串的一个集合，处理这些串将会成功地终止在这个状态。自动机的语言是这些集合的并。由引理 5.5.2，我们可以将任意的有限自动机转化为一个等价的具有单一接收状态的 NFA- λ 。为了简化从有限自动机产生正则表达式的过程，我们假设自动机只具有一个接收状态。

[193]

在节点删除算法之中，将采用 NFA- λ 的状态的序号来标识状态图中的路径。从状态 q_i 到 q_j 的弧的标记标识为 $w_{i,j}$ 。如果状态 q_i 到 q_j 没有弧，那么 $w_{i,j} = \emptyset$ 。

算法 6.2.2 从有限自动机构造正则表达式

输入：有限自动机的状态图 G ，它只含有一个接收状态。

设 q_0 是开始状态， q_i 是接收状态。

1. repeat

1.1. 选择非 q_0 和 q_i 的节点 q_j

1.2. 根据下面的处理过程从 G 中删除节点 q_j

1.2.1. for 每一个不等于 i 的 j, k (包括 $j=k$) do

i) if $w_{j,i} \neq \emptyset, w_{i,k} \neq \emptyset$ 并且 $w_{i,i} = \emptyset$, then 添加一条从节点 q_j 到节点 q_k 的标记为 $w_{j,i}w_{i,k}$ 的弧。

ii) if $w_{j,i} \neq \emptyset, w_{i,k} \neq \emptyset$ 并且 $w_{i,i} \neq \emptyset$, then 添加一条从节点 q_j 到节点 q_k 的标记

为 $w_{j,i}(w_{i,i})^*w_{i,k}$ 的弧。

iii) if 节点 q_j 和 q_k 之间具有标记为 w_1, w_2, \dots, w_s 的弧, then 将这些弧替换为一条标记为 $w_1 \cup w_2 \cup \dots \cup w_s$ 的弧。

1.2.2. 删除节点 q_i 以及 G 中所有连接它的弧。

until G 中只存在节点 q_0 和 q_r ,

2. 决定 G 接收的表达式。

删除节点 q_i 是通过查找所有以 q_i 作为中间节点的长度为 2 的路径 q_j, q_i, q_k 来完成的。添加一条从 q_j 到 q_k 的弧, 但会绕过节点 q_i 。如果没有 q_i 到其自身的弧, 则新的弧上的标记即为原路径上的两条弧上标记的连接。如果 $w_{i,i} \neq \emptyset$, 在从 q_i 进入 q_k 之前, 可以经历 $w_{i,i}$ 任意次。因此新弧的标记为 $w_{j,i}(w_{i,i})^*w_{i,k}$ 。这些图的转换如下所示:



194

在算法的第 2 步中, 可能会出现乞求问题; 整个算法的目的是确定 G 接收的表达式。节点删除过程完成之后, 可以很容易地从结果图之中获得正则表达式。化简之后的图最多只有两个节点, 开始节点和接收节点。如果他们是同一个节点, 则化简之后的图具有如下形式:



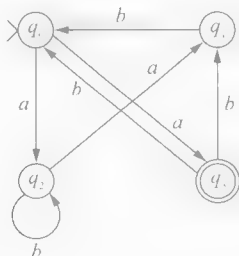
接收 u^* 。开始状态和接收状态不相同的图, 化简之后如下:



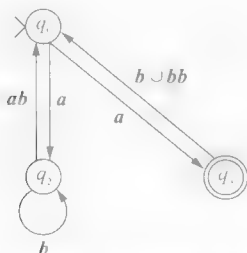
它接收的表达式为: $u^*v(w \cup xu^*v)^*$ 。如果图中具有标识为 \emptyset 的弧, 那么可以简化表达式

算法 6.2.2 可以用来构造具有多个接收状态的有限状态自动机接收的语言。对每一个接收状态, 可以产生一个该状态接收的串的表达式。对每一个接收状态的正则表达式进行简单的求并就能获得自动机的语言。

例 6.2.1 利用算法 6.2.2 的化简方法, 产生 NFA 接收的语言的正则表达式, NFA 的状态图如下:

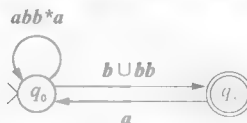


删除节点 q_1 得到:



[195]

删除节点 q_1 产生了从 q_0 到 q_0 的第二条路径, 它标识为从 q_0 到 q_0 的弧上的表达式. 删除 q_2 产生:



它相应的语言是 $(abb^*a)^*(b \cup bb)(a(abb^*a)^*(b \cup bb))^*$. □

前面两节的结果产生的正则语言的特征最初由 Kleene 提出. 6.1 节之中提出的构造方法可以用来构造接收任何正则语言的 NFA-λ. 相反, 算法 6.2.2 产生有限自动机接收的语言的正则表达式. 利用确定型自动机和非确定型自动机的等价性, Kleene 的定理可以用确定型有限自动机接收的语言的形式进行表达.

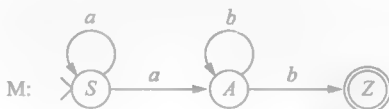
定理 6.2.3 (Kleene) 语言 L 被字母表 Σ 上的 DFA 接收, 当且仅当, L 是 Σ 上的正则表达式.

6.3 正则文法和有限自动机

上下文无关文法称为正则文法, 如果每一个产生式具有下面的形式: $A \rightarrow aB$, $A \rightarrow a$, 或者 $A \rightarrow \lambda$. 在正则文法之中, 可推导的串最多包含一个变量, 如果存在的, 则它是最右端的符号. 一个推导通过引用规则 $A \rightarrow a$ 或者 $A \rightarrow \lambda$ 结束.

语言 a^*b^* 由文法 G 产生, 它被 NFA M 接收, M 的状态为 S , A 和 Z .

$G: S \rightarrow aS \mid aA$
 $A \rightarrow bA \mid b$



下面给出了 M 接收 $aabb$ 的计算, 同时也给出了文法 G 中产生该串的推导过程.

推导	计算	处理的字符串
$S \Rightarrow aS$	$[S, aabb] \vdash [S, abb]$	a
$\Rightarrow aaA$	$\vdash [A, bb]$	aa
$\Rightarrow aabA$	$\vdash [A, b]$	aab
$\Rightarrow aabb$	$\vdash [Z, \lambda]$	$aabb$

[196]

自动机的计算以输入串开始, 然后连续地处理最左边的符号, 当整个串处理完时则停止. 另一方面, 推导从文法的开始符号开始, 并将终结符号加到推导的句子中作为前缀. 通过应用空规则或者右边只有一个终结符的规则终止推导.

上面的例子说明了正则文法产生终结字符串和自动机的计算处理串的过程的对应关系. 自动机的状态和推导串中的变量是一样的. 当整个串处理完毕时, 计算终止; 同时, 结果由终结状态指出. 它不与文法之中的任何变量相对应的接收状态 Z 会被添加到 M 中, 用于表示 G 的推导完成.

NFA M 的状态图可以直接从正则文法的规则中构造出来. 自动机的状态包括文法中的变量或者可能还有一个额外的接收状态. 在前面的例子中, M 中的转换 $\delta(S, a) = S$, $\delta(S, a) = A$ 和 $\delta(A, b) = A$.

分别同 G 中的规则 $S \rightarrow aS$ 、 $S \rightarrow aA$ 和 $A \rightarrow bA$ 相对应。规则的左边表示了自动机当前的状态，右边的终结符是输入符号。与右边变量相对应的状态是作为转换结果而进入的状态。

因为终止一个推导的规则并不添加新的变量到串中，所以应用 λ 规则和 $A \rightarrow a$ 规则的结果必须结合到相应的自动机的构造之中。

定理 6.3.1 设 $G = (V, \Sigma, P, S)$ 是正则文法。定义 NFA $M = (Q, \Sigma, \delta, S, F)$ 如下：

- i) $Q = \begin{cases} V \cup \{Z\} & \text{其中 } Z \notin V, \text{ 如果 } P \text{ 包含规则 } A \rightarrow a \\ V & \text{其他} \end{cases}$
- ii) $\delta(A, a) = B$ 只要 $A \rightarrow aB \in P$
 $\delta(A, a) = Z$ 只要 $A \rightarrow a \in P$
- iii) $F = \begin{cases} \{A \mid A \rightarrow \lambda \in P\} \cup \{Z\}, & \text{如果 } Z \in Q, \\ \{A \mid A \rightarrow \lambda \in P\}, & \text{其他} \end{cases}$

那么 $L(M) = L(G)$

证明：来自文法的规则构造自动机的转换允许在 M 的计算中跟踪 G 的推导。终结字符串的推导具有下面的形式： $S \Rightarrow \lambda$ 、 $S \Rightarrow wC \Rightarrow wa$ 或 $S \Rightarrow wC \Rightarrow w$ ，其中推导 $S \Rightarrow wC$ 包含了形式为 $A \rightarrow aB$ 的规则的应用。可以用归纳来证明 M 之中存在这样的计算，它处理串 w 并终止在状态 C ，只要 G 中具有形式为 wC 的句子（练习6）。

[197]

首先，我们将证明 G 产生的每一个串都能被 M 接收。如果 $L(G)$ 包含空串，那么 S 是 M 的一个接收状态，并且 $\lambda \in L(M)$ 。通过应用规则 $C \rightarrow a$ 或者 $C \rightarrow \lambda$ 终止空串的推导。在形式为 $S \Rightarrow wC \Rightarrow wa$ 的推导之中，最后应用的规则对应于 $\delta(C, a) = Z$ ，它会使自动机终止在接收状态 Z 。形式为 $S \Rightarrow wC \Rightarrow w$ 的推导，通过应用 λ 规则来终止。因为 $C \rightarrow \lambda$ 是 G 中的规则，所以状态 C 是 M 的接收状态。在 M 中 w 的接收对应于推导 $S \Rightarrow wC$ 。

另一方面，我们必须证明 $L(M) \subseteq L(G)$ 。设 $w = ua$ 是被 M 接收的串，接收 w 的计算具有如下的形式：

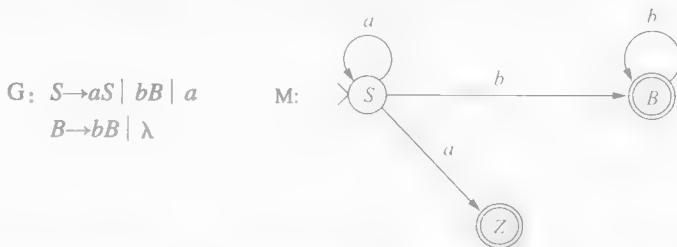
$$[S, w] \vdash [B, \lambda], \text{ 其中 } B \neq Z,$$

或者

$$[S, w] \vdash [A, a] \vdash [Z, \lambda].$$

在前一种情况下， B 是 G 的一个 λ 规则的左边变量。串 wB 可以通过应用与转换相对应的规则推导出来。 w 的产生通过应用 λ 规则完成。类似地， uA 的推导可以从与计算 $[S, w] \vdash [A, a]$ 中的转换相对应的规则推导而来，通过应用规则 $A \rightarrow a$ 终止推导而获得串 w 。因此， M 接收的每一个串都在 G 的语言中。 ■

例 6.3.1 文法 G 产生的和 NFA M 接收的语言为 $a^*(a \cup b^+)$ 。



□

前面的转换可以反过来从 NFA 构造正则文法。转换 $\delta(A, a) = B$ 产生规则 $A \rightarrow aB$ 。因为每一个转换会产生一个新的机器状态，因此不会产生形式为 $A \rightarrow a$ 的规则。由转换得到的规则产生形式为 $S \Rightarrow wC$ 的推导，它是自动机中的仿真计算。必须添加规则来结束推导。当 C 是一个接收状态时，终止在状态 C 的计算即为 w 的接收过程。通过应用规则 $C \rightarrow \lambda$ 来完成推导 $S \Rightarrow wC$ ， G 中的这个推导将产生 w 。在文法之中，为自动机的每一个接收状态增加一个 λ 规则，这样文法便完成了一定理 6.3.2 证明这个非形式化的方法是正确的。形式化的证明将留做练习。 □

[198]

定理 6.3.2 $\text{NFAM} = (Q, \Sigma, \delta, q_0, F)$ 。定义正则文法 $G = (V, \Sigma, P, q_0)$ 如下:

- i) $V = Q$,
- ii) $q_i \rightarrow aq_j \in P$ 只要 $\delta(q_i, a) = q_j$,
- iii) $q_i \rightarrow \lambda \in P$ 如果 $q_i \in F$ 。

那么 $L(G) = L(M)$ 。

定理 6.3.1 和定理 6.3.2 简述了构造过程, 依次地应用这两个定理可以将一个自动机转化为文法, 然后再转换回来。从一个 NFA M 开始, 转换序列具有下面的形式:

$$M \longrightarrow G \longrightarrow M'.$$

因为 G 只包含形式为 $A \rightarrow aB$ 或 $A \rightarrow \lambda$ 的规则, 因此 NFA M' 和 M 是一样的。

正则文法可以转化为一个 NFA, 依次地, 可以再被转化为文法 G'

$$G \longrightarrow M \longrightarrow G'.$$

通过增加一个简单得变量 (称作 Z) 和规则 $Z \rightarrow \lambda$ 到文法之中, 这些转换得到的文法 G' 可以直接从 G 中获得。所有形式为 $A \rightarrow a$ 的规则可以用 $A \rightarrow aZ$ 替换。

例 6.3.2 接收 $L(M)$ 的正则文法 G' 可以从例 6.3.1 的自动机 M 构造而得,

$$\begin{aligned} G': S &\rightarrow aS \mid bB \mid aZ \\ B &\rightarrow bB \mid \lambda \\ Z &\rightarrow \lambda \end{aligned}$$

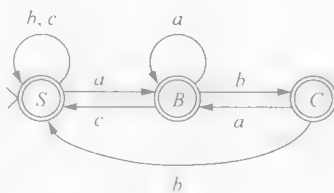
转换提供了 S 规则和第一个 B 规则。因为 B 和 Z 是接收状态, 所以 λ 规则会被加到文法中。 □

两个转换使得我们能够得出结论: 正则文法产生的语言正好能够被有限自动机接收。紧接定理 6.2.3 和定理 6.3.1 (由正则文法产生的语言) 的是正则集。从正则文法到自动机的转换保证了每一个正则集合能够由某个正则文法产生。这就得到了 3.3 节中提出的正则语言的特征: 它是正则文法产生的语言。

[199]

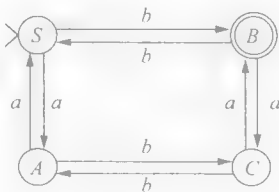
例 6.3.3 例 3.2.12 中的正则文法的语言为 $\{a, b, c\}$ 上的不包含 abc 子串的字符串的集合。利用定理 6.3.1 构造一个接收该语言的 NFA

$$\begin{aligned} S &\rightarrow bS \mid cS \mid aB \mid \lambda \\ B &\rightarrow aB \mid cS \mid bC \mid \lambda \\ C &\rightarrow aB \mid bS \mid \lambda \end{aligned}$$



□

例 6.3.4 从例 5.3.5 中的 DFA 可以构造一个字母表为 $\{a, b\}$ 的正则文法, 该文法产生含有偶数个 a 和奇数个 b 的字符串。下面重新产生这个自动机, 其中状态 $[e_a, e_b]$ 、 $[o_a, e_b]$ 、 $[e_a, o_b]$ 和 $[o_a, o_b]$ 分别重命名为 S 、 A 、 B 和 C 。



相应的文法为:

$$\begin{aligned} S &\rightarrow aA \mid bB \\ A &\rightarrow aS \mid bC \\ B &\rightarrow bS \mid aC \mid \lambda \\ C &\rightarrow aB \mid bA. \end{aligned}$$

□

6.4 正则语言的封闭性质

我们已经给出了正则语言的定义,并且知道如何产生和接收正则语言。字母表 Σ 上的语言是正则的,如果它是:

- i) Σ 上的一个正则集合;
- ii) 能够被一个 DFA、NFA 或者 NFA- λ 接收;
- iii) 由一个正则文法产生。

一个语言族在某个操作下是封闭的,如果应用该操作到该族的成员上时产生该族的另一个成员。每一个正则的等价规则将用来展示正则语言族的封闭性质。

正则集合的递归定义确立了正则集合对一元 Kleene 星和二元操作并和连接的封闭性。这已经在定理 5.5.3 之中用有限状态自动机的接收进行了证明。

定理 6.4.1 设 L_1 和 L_2 是两个正则语言。则,语言 $L_1 \cup L_2$ 、 $L_1 \cap L_2$ 和 L_1^* 都是正则语言。

正则语言对补运算是封闭的。如果 L 是 Σ 上的正则语言,那么 $\bar{L} = \Sigma^* - L$, 该集合包含 Σ^* 中所有不在 L 中的串。定理 5.3.3 利用了 DFA 的这个性质从一个接收 L 的自动机构造了一个接收 \bar{L} 的自动机。补和并相结合可以推出正则语言对交是封闭的。

定理 6.4.2 设 L 是 Σ 上的正则语言,那么语言 \bar{L} 也是正则的。

定理 6.4.3 这 L_1 和 L_2 是 Σ 上的正则语言,那么语言 $L_1 \cap L_2$ 也是正则语言。

证明: 由德摩根定律:

$$L_1 \cap L_2 = \overline{(\bar{L}_1 \cup \bar{L}_2)}.$$

等式的右边是正则的,因为它从 L_1 和 L_2 通过补和并运算获得的。 ■

封闭性为我们确定语言的规律提供了又一个工具。补和交操作,以及并、连接和 Kleene 星操作,使我们组合正则语言的时候能够保持正则性。

例 6.4.1 设 L 是 $\{a, b\}$ 上的语言,它包含所有含有子串 aa 但不含子串 bb 的字符串。正则语言 $L_1 = (a \cup b)^* aa (a \cup b)^*$ 和 $L_2 = (a \cup b)^* bb (a \cup b)^*$ 分别是含有子串 aa 和子串 bb 的语言,因此 $L = L_1 \cap \bar{L}_2$ 是正则的。 □

例 6.4.2 设 L 是 $\{a, b\}$ 上任意的正则语言。语言

$$L_1 = \{u \mid u \in L \text{ 并且 } u \text{ 恰好含有一个 } a\}$$

是正则的。正则表达式 $b^* ab^*$ 描述了正好包含一个 a 的字符串的集合。因为正则语言的交是封闭的,所以语言 $L_1 = L \cap b^* ab^*$ 也是正则的。 □

下面的例子将展示正则语言族的鲁棒性。添加或者删除一个小的数目,事实上可以是任意有限个数字的串,不会将一个正则语言转化为一个非正则的语言。

例 6.4.3 设 L_1 是 Σ 上的正则语言,设 $L_2 \subseteq \Sigma^*$ 是任意的串的有限集合。那么 $L_1 \cup L_2$ 和 $L_1 - L_2$ 都是正则的。根据观察,这里的关键是证明任何有限语言都是正则的。为什么? $L_1 \cup L_2$ 和 $L_1 - L_2$ 的正则性可以由正则语言在并和集合差(练习 8)上的封闭性得到。 □

例 6.4.4 集合 $\text{SUF}(L) = \{v \mid uv \in L\}$ 包含语言 L 所有串的所有后缀。例如,如果 $aabb \in L$, 那么 λ 、 b 、 bb 、 abb 和 $aabb$ 都在 $\text{SUF}(L)$ 之中。下面将证明,如果 L 是正则的,那么 $\text{SUF}(L)$ 也是正则的。因为 L 是正则的,所以存在一个定义 L 的正则表达式,它能够被一个有限状态自动机接收,并且由一个正则文法产生。我们可以采用任何一类正则性来显示 $\text{SUF}(L)$ 也是正则的。

采用文法分类,我们知道 L 由一个正则文法 $G = (V, \Sigma, P, S)$ 产生。假设 G 中没有无用符号。如果含有,可以采用算法 4.4 将无用符号删除,同时文法保持其正则性不变。

G 中 v 的后缀由下面的推导形式产生:

$$S \Rightarrow^* uA \Rightarrow^* uv.$$

直觉地,添加规则 $S \rightarrow A$ 到 G 中将直接产生后缀

$$S \Rightarrow^* A \Rightarrow^* v.$$

不幸地是,结果文法将不再是正则的。为了解决这个问题,我们将采用第 4 章的文法转换。

我们重新定义新的文法 $G' = (V', \Sigma', P', S')$, 其中

$$V' = V \cup \{S'\}$$

$$P' = P \cup \{S' \rightarrow A \mid A \in V\}.$$

[202]

G' 中的推导只使用一个不在 G 中的规则。任何 L 中的串都能够通过下面的形式产生:

$$S' \Rightarrow S \Rightarrow w,$$

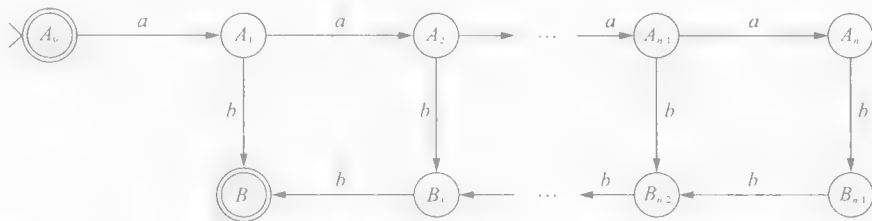
同时, 余下的后缀串通过下面的产生式产生:

$$S' \Rightarrow A \Rightarrow w.$$

因此, $L(G') = \text{SUF}(L)$ 。删除 G' 中的 λ 规则和链规则可以获得一个等价的正则文法。 \square

6.5 非正则语言

非完整定义的 DFA 如下,



它接收的语言为 $\{a^i b \mid i \leq n\}$ 。状态 A_i 记录输入串中前导 a 的个数。一旦处理到第一个 b , 则自动机进入标记为 B_i 的状态。当处理到相同数目的 b 时, 进入接收状态 B_0 。可以扩展这个策略来接收语言 $L = \{a^i b \mid i \geq 0\}$, 因为它需要无限多个状态。然而, 这里可能具有其他的只要求有限个状态的自动机接收 L 。我们将说明, 此处是例外的, L 不能被任何 DFA 接收, 因此它不是一个正则语言。

采用反证法来证明语言 $L = \{a^i b \mid i \geq 0\}$ 不是正则的。假设存在接收 L 的 DFA, 下面将推导出它必然具有与 A_1, A_2, \dots 相同形式的用于记录 a 的个数的状态。这导致了自动机必须具有无限多个状态, 这与假设 DFA 只有有限个状态矛盾。因此我们能够得出结论, 不存在接收 L 的 DFA。

我们开始假设 L 能够被某个 DFA 接收, 并称其为 M 。我们将采用扩展转换函数 $\hat{\delta}$ 来展示自动机 M 必须具有无限多个状态。设 A_i 是机器处理串 a^i 进入的状态, 即 $\hat{\delta}(q_0, a^i) = A_i$ 。对所有的 $i, j \geq 0$ 且 $i \neq j$, 都有 $a^i b' \in L$ 且 $a^j b' \notin L$ 。因为前者是接收状态, 后者是拒绝状态, 所以 $\hat{\delta}(q_0, a^i b') \neq \hat{\delta}(q_0, a^j b')$ 。现在:

$$\hat{\delta}(q_0, a^i b') = \hat{\delta}(\hat{\delta}(q_0, a^i), b') = \hat{\delta}(A_i, b') \in L$$

[203]

和

$$\hat{\delta}(q_0, a^j b') = \hat{\delta}(\hat{\delta}(q_0, a^j), b') = \hat{\delta}(A_j, b') \notin L.$$

因此, $\hat{\delta}(A_i, b') \neq \hat{\delta}(A_j, b')$ 。在确定型自动机之中, 处于相同状态的两个计算, 处理完同一个串之后必须终止在同一个状态。因为计算 $\hat{\delta}(A_i, b')$ 和 $\hat{\delta}(A_j, b')$ 处理同样的串, 并终止于不同的状态, 所以我们可以得出结论 $A_i \neq A_j$ 。

我们已经展示了状态 A_i 和 A_j 对所有的 $i \neq j$ 是不相同的。任何接收 L 的确定型有限状态自动机都必须含有无限个对应于 A_0, A_1, A_2, \dots 的状态。这与 DFA 具有有限个状态的条件矛盾。因此, 没有接收 L 的 DFA, 或者等价地, L 不是正则的。前面的讨论证明了定理 6.5.1 的正确性。

定理 6.5.1 语言 $\{a^i b' \mid i \geq 0\}$ 不是正则的。

定理 6.5.1 的证明是一种不存性证明的例子。我们显示, 无论设计者多么聪明, 都不能构造接收语言 $\{a^i b' \mid i, j \geq 0\}$ 的 DFA。存在证明和不存在证明具有本质的不同。可以通过构造接收语言的自动机来证明语言是正则的。证明不存在要求证明没有接收该语言的机器。定理 6.5.1 可以用来确定一系列的語言的非正则性。

推论 6.5.2 (为了证明定理 6.5.1) 设 L 是 Σ 上的语言。如果存在一序列的不同串 $u_i \in \Sigma^*$ 和 v_i

$\in \Sigma^*$, $i \geq 0$, 使得 $u_i v_i \in L$, $u_i v_j \notin L$, $i \neq j$ 。那么 L 不是正则语言。

证明和定理 6.5.1 一样, 将 u_i 替换为 a^i , v_i 替换为 b^i 。

例 6.5.1 $\{a, b\}$ 上的回文串的集合 L 不是正则的。由推论 6.5.2, 能够找到两个满足推论条件的串序列 u_i, v_i 。这两个串是:

$$u_i = a^i b$$

$$v_i = a^i$$

204 它们完全满足要求。 □

例 6.5.2 文法是作为一个定义语言语法的形式化结构而引入的。推论 6.5.2 可以用来显示规则文法并不是一个定义程序语言的足够强大的工具, 如果这个程序语言包含前缀形式的算术和布尔表达式。文法 AE

$$\begin{aligned} \text{AE: } S &\rightarrow A \\ A &\rightarrow T \mid A + T \\ T &\rightarrow b \mid (A) \end{aligned}$$

用 $+$ 号、括号和操作数 b 生成加法表达式。例如 (b) 、 $b + (b)$ 和 $((b))$ 都在 $L(\text{AE})$ 中。

中缀记号允许嵌套的括号, 这事实上是要求。推导

$$\begin{aligned} S &\Rightarrow T \\ &\Rightarrow (A) \\ &\Rightarrow (T) \\ &\Rightarrow (b) \end{aligned}$$

展示了利用规则 AE 产生串 (b) 的过程。在应用 $T \rightarrow b$ 终止推导之前, 重复应用规则序列 $T \Rightarrow (A) \Rightarrow (T)$ 会得到串 $((b))$, $((((b))))$, ...。串 (b) 和 $((b))$ 满足推论 5.6.2 中要求的序列 u_i 和 v_i 的条件。所以, 文法 AE 定义的语言不是正则的。类似的讨论可以用来展示编程语言如 C、C++ 和 Java 以及其他的一些语言, 都不是正则的。 □

正如正则语言的封闭性可以用来确定正则性, 它们也可以用来展示语言的非正则性。

例 6.5.3 语言 $L = \{a^i b^j \mid i, j \geq 0, \text{ 并且 } i \neq j\}$ 不是正则的。如果 L 是正则的, 根据定理 6.4.2 和定理 6.4.3, $\overline{L} \cap a^* b^*$ 也是正则的。但 $\overline{L} \cap a^* b^* = \{a^i b^j \mid i \geq 0\}$, 而我们已经知道了这个语言不是正则的。 □

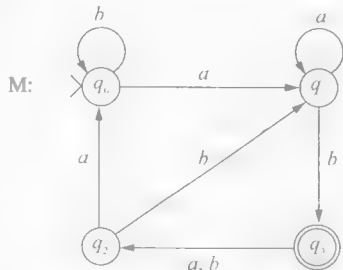
6.6 规则语言的泵引理

在前面一节中, 存在的非正则语言是通过展示不能构造一个接收该语言的 DFA 来证明的。在本节中, 将引入一个确定非正则性的更加通用的准则——主要结果, 正则表达式的泵引理, 要求正则语言中的串能够允许分开来满足某种重复的性质。

泵作用一个串是通过重复地从原始的串中抽取子串来构造新的串。DFA 状态图中的接收表示了抽取串的过程。考虑 $L(M)$ 中的串 $z = ababbbaaab$, 它可以分解成子串 u, v 和 w 其中 $u = a, v = bab, w = baaab$ 和 $z = uvw$ 。串 $a(bab)^i baaab$ 可以通过泵作用于串 $ababbbaaab$ 中的子串 bab 而获得。

通常, 在 DFA M 之中处理 z 相应地会在 M 的状态图中产生一条路径。 z 分解成 u, v 和 w 会在状态图中将路径打破成为三条子路径。子串 $u = a$ 和 $w = baaab$ 的计算产生的子路径分别为 q_0, q_1 和 $q_1, q_3, q_2, q_0, q_1, q_3$ 。处理分解的第二个部分产生环 q_1, q_3, q_2, q_1 。泵作用产生的 $uv^i w$ 同样能够被 DFA 接收, 因为子串 v 的重复只是在处理 w 并终止在状态 q_3 之前, 简单地增加额外的周游回路 q_1, q_3, q_2, q_1 。

泵引理要求对 DFA 的语言中所有足够长的串存在这样的一个分解。下面将列出两个引理, 它们



205

给出了在 DFA 状态图的路径中存在回路的条件。将采用鸽巢原理进行证明,该原理是基于对给定的盒子的数目,以及一个数量大于盒子数目的需要分发到这些盒子之中的物品的观察,观察发现至少有一个盒子中物品的数量大于或等于2。

引理 6.6.1 设 G 是具有 k 个状态的 DFA 的状态图 G 中任何一条长度为 k 的路径中必包含一个环。

证明: 长度为 k 的路径包含 $k+1$ 个节点,因为 G 中只有 k 个节点,所以必定有一个节点在路径之中至少出现了两次,记为 q_i 。从第一次出现的 q_i 到第二次出现的 q_i 的子路径就是所要求的环。

长度大于 k 的路径可以分为一个初始的长度为 k 的子路径和余下的路径。引理 6.6.1 保证了在初始的子路径之中具有一个环。推论 6.6.2 形式化表达了前面的叙述。

206

推论 6.6.2 设 G 是具有 k 个状态的 DFA 的状态图, p 是长度大于等于 k 的路径。路径 p 可以分解为子路径 q 、 r 和 s , 其中 $p = qrs$ 、 qr 的长度小于或者等于 k , 并且 r 是一个环。

定理 6.6.3 (正则语言的泵引理) 设 L 是正则语言, 它被具有 k 个状态的 DFA M 接收。设 z 是 L 的任意长度大于等于 k 的串, 那么, z 可以写成 uvw 的形式, 其中 $\text{length}(uv) \leq k$, $\text{length}(v) > 0$ 并且对于所有的 $i \geq 0$ 有 $uv^i w \in L$ 。

证明: 设 $z \in L$, 它的长度为 n , $n \geq k$ 。在 M 处理 z 的时候, 将在 M 的状态图中产生一条长度为 n 的路径。根据推论 6.6.2, 这条路径可以分解为子路径 q 、 r 和 s , 其中 r 是状态图中的一个环。将 z 分解成 u 、 v 和 w , 它们分别是路径 q 、 r 和 s 上的标记。

串 $uv^i w$ 所对应的路径与串 uvw 开始和终止节点相同。惟一的不同是环路 r 出现的次数。因此, 如果 uvw 能够被 M 接收, 那么 $uv^i w$ 也能够被 M 接收。

在证明泵引理的过程中, 并没有特别指出接收 L 的特定 DFA 的性质。讨论对于所有的 DFA 都成立, 包括只具有最小数目状态的 DFA。陈述定理的时候可以将 k 加强是接收 L 的最小 DFA 的状态数。

泵引理是证明一个语言不是正则语言的强大的工具。正则语言中的每一个长度大于等于 k 的串, 其中 k 的值由特定的泵引理确定, 都必须具有一个适当的分解。为了说明语言不是正则的, 只需要找到一个不满足泵引理条件的串即可。使用泵引理确认非正则性将在下面的例子之中展示。它包含选择一个 L 的串 z , 并说明不存在 z 的一个分解 uvw , 使得对所有的 $i \geq 0$ 都有 $uv^i w \in L$ 。

例 6.6.1 设 $L = \{z \in \{a\}^* \mid \text{length}(z) \text{ 是平方数} \}$ 。假设 L 是正则的。这表明 L 能够被某个 DFA 接收。设该 DFA 的状态数目是 k 。由泵引理, 任何一个长度大于等于 k 的串 $z \in L$ 都可以分解成子串 u 、 v 和 w , 从而使得 $\text{length}(uv) \leq k$, $v \neq \lambda$ 并且对所有点 $i \geq 0$, $uv^i w \in L$ 。

考虑串 $z = a^k$ 的长度为 k^2 。因为 $z \in L$ 中, 并且长度大于 k , 因此 z 可以写成 $z = uvw$ 其中 u 、 v 和 w 满足泵引理的条件。特别地, $0 < \text{length}(v) \leq k$ 。这点可以用来确定 $uv^2 w$ 长度的上限。

207

$$\begin{aligned} \text{length}(uv^2 w) &= \text{length}(uvw) + \text{length}(v) \\ &= k^2 + \text{length}(v) \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \\ &= (k+1)^2. \end{aligned}$$

$uv^2 w$ 的长度大于 k^2 而小于 $(k+1)^2$, 因此不是一个平方数, 因此, 串 $uv^2 w$ 不在 L 之中。我们已经展示了不存在 z 的一个满足泵引理的分解, 这与假设 L 是正则的矛盾, 因此 L 不是正则的。

例 6.6.2 说明语言 $L = \{a^i \mid i \text{ 是质数} \}$ 不是正则的。假设存在一个具有 k 个状态的接收 L 的 DFA。设 n 是大于 k 的最小的质数。由泵引理可得, a^n 可以分解成子串 uvw , $v \neq \lambda$ 并且对所有 $i \geq 0$, $uv^i w \in L$ 。假设存在这样的分解。

如果 $uv^{n+1} w \in L$, 那么它的长度一定是质数, 但是:

$$\begin{aligned} \text{length}(uv^{n+1} w) &= \text{length}(uv^n w) \\ &= \text{length}(uvw) + \text{length}(v^n) \\ &= n + n(\text{length}(v)) \\ &= n(1 + \text{length}(v)). \end{aligned}$$

因为长度不是质数, 所以 $uv^{n+1} w$ 不在 L 中。因此不能够把 a^n 分解为满足泵引理的 uvw , 于是可以得

出结论, L 不是正则的。 \square

在前面的例子中, 串的长度的约束对于证明语言不是正则的是充分的。通常可以用串元素之间的数值关系来说明不存在一个满足泵引理的子串。下面将提出另一个论据, 这次我们用泵引理来说明 $\{a^i b^j \mid i \geq 0\}$ 不是正则的。

208

例 6.6.3 说明 $L = \{a^i b^j \mid i \geq 0\}$ 不是正则的。我们必须找到一个 L 中的适当长度的串, 它不具有满足泵引理的子串。假设 L 是正则的, k 是泵引理中指定的数。设串 $z = a^k b^k$, z 的任意满足泵引理的分解必须具有下面的形式:

$$\begin{array}{ccc} u & v & w \\ a^i & a^j & a^{k-i-j} b^k, \end{array}$$

其中 $i+j \leq k$ 且 $j > 0$ 。泵出来任何下面形式的子串 $uv^2w = a^i a^j a^{k-i-j} b^k = a^k a^j a^k$, 不在 L 中。由于 z 没有满足泵引理的分解, 所有 L 不是正则的。 \square

例 6.6.4 语言 $L = \{a^i b^m c^n \mid 0 < i, 0 < m < n\}$ 不是正则的。假设 L 能够被 k 个状态的 DFA 接收。那么, 由泵引理, 每一个 L 中长度大于等于 k 的串 z 都可以写成 $z = uvw$, 其中 $\text{length}(uv) \leq k$, $\text{length}(v) > 0$, 并且对所有的 $i \geq 0$ 有 $uv^i w \in L$ 。

考虑串 $z = ab^k c^{k+1}$, 它属于 L 。我们必须说明不存在 z 的满足泵引理的适当分解。 z 的任何分解都只可能具有两种形式之一, 分情况进行讨论:

情况 1: 一个 $a \in v$ 的分解具有下面的形式:

$$\begin{array}{ccc} u & v & w \\ ab^i & b^j & b^{k-i-j} c^{k+1} \end{array}$$

其中 $i+j \leq k-1$ 且 $j > 0$ 。泵作用 v , 产生 $uv^2w = a^i b^i b^j b^{k-i-j} c^{k+1} = ab^k b^j c^{k+1}$, 它不在 L 中。

情况 2: 一个 $a \in v$ 的分解具有下面的形式:

$$\begin{array}{ccc} u & v & w \\ \lambda & ab^i & b^{k-i} c^{k+1} \end{array}$$

其中 $i \leq k-1$ 。泵 v 零次, 得到 $uv^0w = b^{k-i} c^{k+1}$, 它不在 L 中, 因为它不包含 a 。

由上面的分析可知, $ab^k c^{k+1}$ 不具有一个可泵作用的子串, 所以 L 不是正则的。 \square

泵引理可以用来确定一个 DFA 接收的语言的大小。泵作用在一个串上, 将产生一个无限序列的串, 它们都能够被 DFA 接收。为了确定一个正则语言是有限的还是无限的, 只需要确定它是否包含一个可泵的串。

定理 6.6.4 设 M 是 k 个状态的 DFA。

i) $L(M)$ 是非空的, 当且仅当 M 接收一个长度小于 k 的串。

ii) $L(M)$ 具有无限个元素, 当且仅当 M 接收串 z , 并且 $k \leq \text{length}(z) < 2k$ 。

证明: i) 如果 M 接收一个长度小于 k 的串, $L(M)$ 显然是非空的。

现在设 M 是语言非空的自动机, 并且 z 是 $L(M)$ 中的最小的串。假设 z 的长度大于 $k-1$ 。由泵引理, z 可以写成 uvw , 其中 $uv^i w \in L$ 。特别地, $uv^0w = uw$ 是 L 中长度小于 z 的串。这与假设 z 是长度最小的串矛盾。因此, $\text{length}(z) < k$ 。

ii) 如果 M 接收串 z , $k \leq \text{length}(z) < 2k$, 那么 z 可以写成 uvw , 其中 u, v 和 w 满足泵引理的条件。这表明对所有 $i \geq 0$ 有 $uv^i w \in L$ 。

假设 $L(M)$ 是无限的。我们必须说明 $L(M)$ 中存在长度在 k 和 $2k-1$ 之间的串。因为在有限字母表上只有有限个串的长度小于 k , $L(M)$ 中一定包含长度大于 $k-1$ 的串。选择 $L(M)$ 中长度大于 $k-1$ 的最小的串 z 。如果 $k \leq \text{length}(z) < 2k$, 那么结论成立。假设 $\text{length}(z) \geq 2k$, 根据泵引理可知 $z = uvw$, 其中 $\text{length}(v) \leq k$, 并且 $uv^0w = uw \in L(M)$ 。因为 uw 是一个长度大于 $k-1$ 的串, 而其长度却严格的小于 z 的长度, 与假设矛盾。 \blacksquare

前面的结果给出了一个确定 DFA 的语言的基数的方法。如果 k 是自动机的状态数, j 是字母表的大小, 那么具有 $(j^k - 1)/(j - 1)$ 个长度小于 k 的串。由定理 6.6.4, 检查所有的这些串将确定自动机的语言是否为空。检查所有长度在 k 和 $2k-1$ 之间的串可以确定语言是有限的还是无限的。当然, 这是

209

一个非常低效的过程。不过，它能够有效地得出下面的推论。

推论 6.6.5 设 M 是 DFA，存在一个算法来确定 $L(M)$ 是空的，有限的或者无限的。

结合正则语言的封闭性和推论 6.6.5，得到一个确定两个 DFA 是否接收相同的语言的方法。

推论 6.6.6 设 M_1 和 M_2 都是 DFA。存在一个确定 M_1 和 M_2 是否等价的算法。

证明：设 L_1 和 L_2 分别是 M_1 和 M_2 接收的语言。根据定理 6.4.1、定理 6.4.2 和定理 6.4.3，语言

$$L = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

是正则的。L 是空的，当且仅当 L_1 和 L_2 是一样的。根据推论 6.6.5，存在一个确定 L 是否为空的算法，或者等价地，确定 M_1 和 M_2 是否接收相同的语言。 ■

210

6.7 Myhill-Nerode 定理

Kleene 定理确定了正则语言和有限自动机之间的关系。在本节中，正则性的特点在于一个语言的串之间存在等价关系。这个特征提供了一个获得接收正则语言的最小的 DFA 的方法，同时也为算法 5.7.2 给出的 DFA 的最小化提供了证明。

定义 6.7.1 设 L 是 Σ 上的语言，如果对所有的 $w \in \Sigma^*$ ， uw 和 vw 要么都在 L 中，要么都不在 L 中，那么串 $u, v \in \Sigma^*$ 在 L 中是不可区分的。

利用是否属于 L 作为区分不同的串的准则，如果对于某些串 w ，它连接到 u 和 v 时将产生在 L 中具有不同的成员关系的串，那么 u 和 v 是可区分的。即 w 区分 u 和 v 当且仅当 uw 和 vw 一个在 L 中而另一个不在。

在 L 中的不可区分性定义了一个 Σ^* 上的二元关系 \equiv_L ；如果 u 和 v 是不可区分的，则 $u \equiv_L v$ 。简单地说， \equiv_L 是自反的、对称的和传递的。这些观察为引理 6.7.2 提供了基础。

引理 6.7.2 对任意的语言 L ，关系 \equiv_L 是一个等价关系。

例 6.7.1 设 L 是正则语言 $a(a \cup b)(bb)^*$ 。串 aa 和 ab 是不可区分的，因为对于任意的 w ， aaw 和 abw 要么都在 L 中，要么都不在 L 中。在前一种情况中， w 中包含偶数个 b ，而后一种情况中， w 为任意其他的串。 b 和 ba 在 L 中同样是不可区分的，因为对任意的 w ， bw 和 baw 都不在 L 中。 a 和 ab 在 L 中是可区分的，因为连接 bb 到 a 将会产生 $abb \notin L$ ，而连接 bb 到 ab 则产生 $abbb \in L$ 。

\equiv_L 的等价类是

代表元素	等价类
$[\lambda]_{\equiv_L}$	λ
$[b]_{\equiv_L}$	$b(a \cup a)^* \cup a(a \cup b)(bb)^* a(a \cup b)^* \cup a(a \cup b)(bb)^* ba(a \cup$
$[a]_{\equiv_L}$	$b)^*$
$[aa]_{\equiv_L}$	a
$[aab]_{\equiv_L}$	$a(a \cup b)(bb)^*$
$[aab]_{\equiv_L}$	$a(a \cup b)b(bb)^*$

□

211

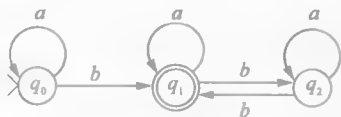
例 6.7.2 设 L 是语言 $\{a^i b^j \mid i \geq 0, j \geq 0, a^i \text{ 和 } a^j, \text{ 其中 } i \neq j, \text{ 在 } L \text{ 中是可区分的}\}$ 。连接 b' 得到 $a^i b^j \in L$ ，而 $a^i b^j \notin L$ 。对于每一个 $a^i, i = 0, 1, 2, \dots$ ，是一个不同的等价类。这个例子说明了，不可区分关系 \equiv_L 可能产生无限个等价类。 □

等价关系 \equiv_L 定义的不可区分性是基于语言的所属关系。我们现在定义一个基于 DFA 计算的串的不可区分性。

定义 6.7.3 设 $M = (Q, \Sigma, \delta, q_0, F)$ 是接收 L 的 DFA。如果 $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ ，那么 $u, v \in \Sigma^*$ 对 M 是不可区分的。

串 u 和 v 对 M 是不可区分的，如果 M 对输入 u 的计算和对输入 v 的计算终止在相同的状态。简单地说，就是这种形式的不可区分性同样定义了 Σ^* 上的等价关系。M 的每个状态 q_i 对于计算是可达的就有相应的一个等价类：所有计算终止在 q_i 的串的集合。所以，DFA M 的等价类的数目最多为状态数目。自动机 M 的不可区分性记做 \equiv_M 。

例 6.7.3 设 M 是接收语言 $a^*ba^*(ba^*ba^*)^*$ 的 DFA, 即具有奇数个 b 的串的集合. 关系 \equiv_M 定义的 Σ^* 上的等价类为:



状态	相关等价类
q_0	a^*
q_1	$a^*ba^*(ba^*ba^*)^*$
q_2	$a^*ba^*ba^*(ba^*ba^*)^*$

□

不可区分关系可以用来提供一个正则性的额外特征. 这些特征利用了不可区分等价关系的右不变性 (right-invariance). 一个 Σ^* 上的等价关系 \equiv 是右不变的, 当 $u \equiv v$, 对任意的 $w \in \Sigma^*$, 能够推出 $uw \equiv vw$. \equiv_L 和 \equiv_M 都是右不变的.

定理 6.7.4 (Myhill-Nerode) 下面的内容是等价的

- i) L 是 Σ 上的正则语言。
- ii) 存在一个 Σ^* 上的右不变的等价关系, 它具有有限个等价类, L 是等价类的一个子集的并。
- iii) \equiv_L 具有有限个等价类。

证明: (i) 推出 (ii). 因为 L 是正则的, 它能够被 DFA $M = (Q, \Sigma, \delta, q_0, F)$ 接收. 我们将说明 \equiv_M 满足条件 (ii). 正如前面提到的, \equiv_M 的等价类的个数最多和 M 的状态数相等. 因此, \equiv_M 的等价类的个数是有限的. 右不变和 M 计算的确定性, 保证了对于任意的 $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ 都有 $\hat{\delta}(q_0, uw) = \hat{\delta}(q_0, vw)$.

接着, 需要证明 L 是 \equiv_M 的一些等价类的并. 对 M 的每一个状态 q_i , 存在一个等价类, 对该类中的串计算终止在状态 q_i . L 是 M 的接收状态相对应的等价类的并。

(ii) 推出 (iii). 设 \equiv 是满足 (ii) 的等价关系, 下面说明每一个 \equiv 的等价类 $[u]$ 是 \equiv_L 的等价类 $[u]_{\equiv_L}$ 的子集。

设 u, v 是 $[u]_{\equiv}$ 中的任意两个串, 即 $u \equiv v$. 根据右不变性, 对任意的 $w \in \Sigma^*$ 都有 $uw \equiv vw$. 所以 uw 和 vw 在同一个等价类中. 因为 L 是 \equiv 的一些等价类的并, 每一个在特定的等价类之中的串, 在 L 中具有相同的所属关系. 所以, uw 和 vw 要么都到 L 中, 要么都不在 L 中. 从而 u 和 v 在 \equiv_L 的同一个等价类之中。

因为对每一个 $u \in \Sigma^*$, $[u]_{\equiv} \subseteq [u]_{\equiv_L}$, 每一个 \equiv_L 的等价类至少包含一个 \equiv 的等价类. 从而推出等价关系 \equiv_L 的等价类的数目不会比 \equiv 的等价类数目多, 因此是有限的。

(iii) 推出 (ii). 为了对此进行证明, 当 \equiv_L 的等价类数目是有限的時候, L 是正则语言, 我们需要构造一个接收 L 的 DFA M_L . M_L 的字母表包含在 L 的符号之中, 并且状态是 \equiv_L 的等价类. 等价类的开始状态包含 λ . 一个等价类是一个接收状态, 如果它包含 $u \in L$. 所有剩下需要做的只是定义转换函数和说明 M_L 的语言是 L .

对每一个 $a \in \Sigma$, 我们定义 $\delta([u]_{\equiv_L}, a) = [ua]_{\equiv_L}$. 由定义, 对于输入字符 a , 从状态 $[u]_{\equiv_L}$ 转换的结果是等价类 $[ua]_{\equiv_L}$. 需要说明的是转换的定义与从等价类中选择哪个特定的元素是不相关的。

设 u 和 v 是两个在 \equiv_L 下等价的串. 对每一个定义良好的转换函数 δ , $[ua]_{\equiv_L}$ 必须和 $[va]_{\equiv_L}$ 是同一个等价类, 或者等价地, $ua \equiv_L va$. 为了确定这一点, 我们需要说明对任意的串 $x \in \Sigma^*$, uax 和 vax 要么同时都在 L 中, 要么同时都不在 L 中. 根据 \equiv_L 的定义, 对任意的 $w \in \Sigma^*$, uw 和 vw 同时在或者同时不在 L 中. 令 $w = ax$ 即可得到结果。

下面需要证明 $L(M_L) = L$. 对任意的串 u , $\hat{\delta}([\lambda]_{\equiv_L}, u) = [u]_{\equiv_L}$. 如果 u 在 L 中, 计算 $\hat{\delta}([\lambda]_{\equiv_L}, u)$ 终止在接收状态 $[u]_{\equiv_L}$. 练习 25 将证明等价类 $[u]_{\equiv_L}$ 的所有元素或者都在 L 中或者都不在 L 中. 所以, 如果 $u \notin L$, 那么 $[u]_{\equiv_L}$ 不是一个接收状态. 从而推出, u 被 M_L 接收当且仅当 $u \in L$.

注意, \equiv_L 的等价类正好是 \equiv_{M_L} 的等价类. 这种 Σ^* 上的不可区分关系由自动机 M_L 产生. ■

例 6.7.4 例 5.7.1 中的 DFA M 接收语言 $(a \cup b)^*(a \cup b)^*$. 关系 \equiv_M 的八个等价类和它们相对应的 M 中的状态如下所示:

212

213

状态	等价类	状态	等价类
q_0	λ	q_4	b
q_1	a	q_5	ba
q_2	aa	q_6	bb^+
q_3	ab^+	q_7	$(aa(a \cup b) \cup ab^+a \cup ba(a \cup b) \cup bb^+a)(a \cup b)^*$

如果对于任意的 w 、 uw 和 vw 或者都在 L 中或者都不在, 那么等价关系 \equiv_L 将 u 和 v 标记为不可区分的。语言 $(a \cup b)(a \cup b)^*$ 的 \equiv_L 等价类如下:

\equiv_L 等价类	
$[\lambda]_{\equiv_L}$	λ
$[a]_{\equiv_L}$	$a \cup b$
$[aa]_{\equiv_L}$	$aa \cup ba$
$[ab]_{\equiv_L}$	$ab^+ \cup bb^+$
$[aba]_{\equiv_L}$	$(aa(a \cup b) \cup ab^+a \cup ba(a \cup b) \cup bb^+a)(a \cup b)^*$

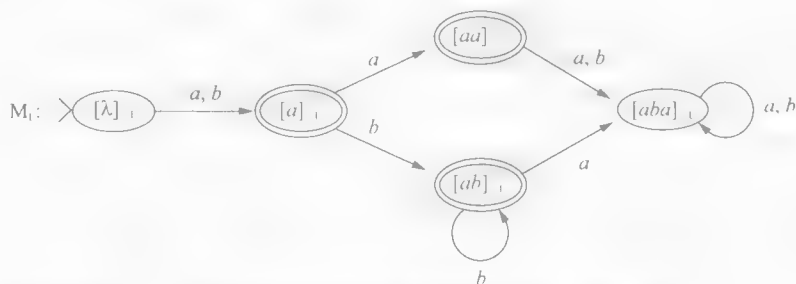
其中, 括号中的串代表等价类中的一个元素。很容易看出, 在同一个等价类中的串是不可区分的, 而不同等价类之中的串是可区分的。

如果我们所记 \equiv_M 的、等价类中的并且计算时终止在状态 q_i 的串为 $cl_M(q_i)$, \equiv_L 和 \equiv_M 的等价类之间的关系如下:

[214]

$$\begin{aligned}
 [\lambda]_{\equiv_L} &= cl_M(q_0) \\
 [a]_{\equiv_L} &= cl_M(q_1) \cup cl_M(q_4) \\
 [aa]_{\equiv_L} &= cl_M(q_2) \cup cl_M(q_5) \\
 [ab]_{\equiv_L} &= cl_M(q_3) \cup cl_M(q_6) \\
 [aba]_{\equiv_L} &= cl_M(q_7).
 \end{aligned}$$

采用 Myhill-Nerode 定理概述的技术, 我们可以从 \equiv_L 的等价类中构造一个接收 L 的 DFA M_L , 得到的 DFA 如下:



它和例 5.7.1 中的通过在 5.7 节中提出的最小化技术获得的 DFA M' 是一样的。□

定理 6.7.5 将说明从 \equiv_L 等价类获得的 DFA M_L 是接收 L 的最小状态的 DFA

定理 6.7.5 设 L 是正则语言, \equiv_L 是由 L 定义的不可区分关系。接收 L 的最小状态的 DFA 是在定理 6.7.4 中指出的根据 \equiv_L 的等价类定义的自动机 M_L 。

证明: 设 $M = (Q, \Sigma, \delta, q_0, F)$ 是任意的接收 L 的 DFA, \equiv_M 是由 M 产生的等价关系。根据 Myhill-Nerode 定理, \equiv_M 的每个等价类是 \equiv_L 的一个等价类的子集。因为 \equiv_M 和 \equiv_L 的等价类都分割 Σ^* , 所以 \equiv_M 的等价类个数至少和 \equiv_L 的一样多。将前面的观察和从 \equiv_L 的等价类构造 M_L 结合起来, 我们能得到:

$$\begin{aligned}
 & \text{M 的状态数} \\
 \geq & \equiv_M \text{ 的等价类数目} \\
 \geq & \equiv_L \text{ 的等价类的数目} \\
 = & M_L \text{ 的状态数。}
 \end{aligned}$$

215

所以, 接收 L 的 DFA M 不可能具有比 M_L 更少的状态数目。我们可以得出结论, M_L 是接收 L 的最小状态数的 DFA。 ■

定理 6.7.5 指出了 M_L 是接收 L 的最小状态数的 DFA。练习 31 将会证明所有接收 L 的最小状态的 DFA 都和 M_L 是一样的, 只是各个状态的名字可能不一样。

定理 6.7.4 和定理 6.7.5 确定了存在惟一的接收 L 的最小状态的 DFA M_L 。最小状态的自动机可以从关系 \equiv_L 的等价类构造。不幸的是, 到此我们还没有给出一个获得等价类的直接方法。定理 6.7.6 说明了状态为 \equiv_L 的等价类的自动机就是在 5.7 节之中最小化算法产生的自动机。

定理 6.7.6 设 M 是接收 L 的 DFA, M' 是从 M 利用 5.7 节最小化构造方法获得的自动机。那么 $M' = M_L$ 。

证明: 根据定理 6.7.5 和练习 31, 如果 M' 的状态数目和 \equiv_L 的等价类的个数相同, 那么 M' 是接收 L 的最小状态数的 DFA。其次由定义 6.7.3, 具有一个等价关系 $\equiv_{M'}$ 将 M' 的每一个状态和一个串的集合相关联。与状态 $[q_i]$ 相对应的 $\equiv_{M'}$ 的等价类如下

$$cl_{M'}([q_i]) = \{u \mid \hat{\delta}'([q_0], u) = [q_i]\} = \bigcup_{q_i \equiv_L [q_i]} \{u \mid \hat{\delta}(q_0, u) = q_i\},$$

其中 $\hat{\delta}'$ 和 $\hat{\delta}$ 分别是 M' 和 M 的扩展转换函数。根据 Myhill-Nerode 定理, $cl_{M'}(q_i)$ 是 $\equiv_{M'}$ 的一个等价类的子集。

假设 M' 的状态数多于 \equiv_L 的等价类的个数, 那么存在 M' 的两个状态 $[q_i]$ 和 $[q_j]$, 使得 $cl_{M'}(q_i)$ 和 $cl_{M'}(q_j)$ 是 \equiv_L 的同一个等价类的子集。这表明, 存在串 u 和 v , 使得 $\hat{\delta}(q_0, u) = q_i$, $\hat{\delta}(q_0, v) = q_j$ 并且 $u \equiv_L v$ 。

因为 $[q_i]$ 和 $[q_j]$ 是 M' 的可区分状态, 所以存在一个区分这两个状态串 w , 即 $\hat{\delta}(q_i, w)$ 被接收而 $\hat{\delta}(q_j, w)$ 不被接收, 或者相反。从而得出, uw 和 vw 在 L 中具有不同的所属关系。这与 $u \equiv_L v$ 表明 uw 和 vw 在 L 中具有同样的所属关系是矛盾的。因此, 假设 M' 状态数多于 \equiv_L 的等价类的个数是错误的。 ■

在 Myhill-Nerode 定理中的正则性特征给出了另一个判断非正则表达式的方法。如果等价关系 \equiv_L 具有无限个等价类, 那么语言 L 不是正则的。

216

例 6.7.5 在例 6.7.2 中, 已经展示了语言 $\{a^i b^j \mid i \geq 0\}$ 具有无限多个 \equiv_L 等价类, 因此它不是正则的。 □

例 6.7.6 利用 Myhill-Nerode 定理来说明语言 $L = \{a^i \mid i \geq 0\}$ 不是正则的。为了完成证明, 首先说明 a^i 和 a^j 在等价关系 \equiv_L 下是可区的, 无论何时都有 $i < j$ 。连接串 a^i 得到 $a^i a^j = a^{i+j} \in L$, 而 $a^j a^i \notin L$, 后者不在 L 中, 因为它的长度比 2^i 大, 但是又小于 2^{i+1} 。所以 a^i 和 a^j 不可区分。这些串将产生一个无穷序列: $[a^0] \equiv_L [a^1] \equiv_L [a^2] \equiv_L [a^4] \equiv_L \dots$, 它们都是 L 的不同的等价类。 □

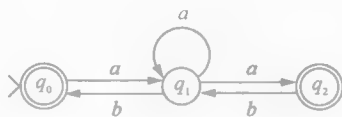
6.8 练习

- 利用 6.2 节的技术构造接收语言 $(ab)^*ba$ 的 NFA- λ 的状态图, 并和练习 5.22 (a) 中构造的 DFA 进行比较。
- 对练习 5.40 的每个状态图, 用算法 6.2.2 为自动机接收的语言构造一个正则表达式。
- 例 5.3.4 中的 DFA M 语言包含 $\{a, b\}$ 上的所有含有偶数个 a 和奇数个 b 的串。利用算法 6.2.2, 为 $L(M)$ 构造一个正则表达式。练习 2.38 要求一个非算法地构造一个该语言的正则表达式, 正如你所见到的, 它是一个多么可怕的任务。
- 设 G 是文法

$$G: S \rightarrow aS \mid bA \mid a$$

$$A \rightarrow aS \mid bA \mid b.$$

- 利用定理 6.3.1 构造一个接收 $L(G)$ 的 NFA M 。
 - 利用 (a) 的结果, 构造一个接收 $L(G)$ 的 DFA M' 。
 - 从 M 构造一个产生 $L(M)$ 的正则文法。
 - 从 M' 构造一个产生 $L(M')$ 的正则文法。
 - 给出 $L(G)$ 的一个正则表达式。
5. 设 M 是如下的 NFA:



217

- 从 M 构造一个产生 $L(M)$ 的正则文法。
 - 给出 $L(M)$ 的正则表达式。
- *6. 设 G 是正则文法, M 是根据定理 6.3.1 从 G 获得的 NFA。证明: 如果 $S \xRightarrow{*} wC$, 那么在 M 中存在计算 $[S, w] \vdash [c, \lambda]$ 。
7. 设 L 是 $\{a, b, c\}$ 上的正则语言, 说明下面的每一个集合都是正则的。
- $\{w \mid w \in L \text{ 并且 } w \text{ 以 } aa \text{ 结尾}\}$ 。
 - $\{w \mid w \in L \text{ 或者 } w \text{ 包含一个 } a\}$ 。
 - $\{w \mid w \notin L \text{ 并且 } w \text{ 不包含 } a\}$ 。
 - $\{uv \mid u \in L \text{ 并且 } v \notin L\}$ 。
8. 证明正则语言族在集合的差的作用下是封闭的。
9. 证明正则语言族和上下文无关语言的交不是封闭的。即是, 如果 L 是正则的, L_1 是上下文无关的, 那么 $L \cap L_1$ 不一定是正则的。
10. 正则语言族在无限并作用下是否是封闭的? 即, 如果 L_0, L_1, L_2, \dots 是正则的, $\bigcup_{i=0}^{\infty} L_i$ 是否一定是正则的? 如果是, 请证明。如果不是, 请给出反例。
- *11. 设 L 是正则语言。说明下面的语言都是正则的。
- 集合 $P = \{u \mid uv \in L\}$ 是 L 中串的前缀。
 - 集合 $L^R = \{w^R \mid w \in L\}$ 是 L 中串的逆。
 - 集合 $E = \{uv \mid v \in L\}$ 中的串在 L 中具有一个后缀。
 - 集合 $SUB = \{v \mid uvw \in L\}$ 中的串是 L 中的串的子串。
- *12. 设 L 是包含长度为偶数的串的正则语言。设 L' 是语言 $\{u \mid uv \in L \text{ 并且 } length(u) = length(v)\}$ 。证明 L' 是正则的。
13. 利用推论 6.5.2 说明下面的每一个集合都不是正则的。
- $\{a, b\}$ 上的具有相同数目的 a 和 b 的串的集合。
 - $\{a, b\}$ 上的长度为偶数的回文串。
 - $\{(,)\}$ 上的串的集合, 这些串中的括号是成对的。例如 $\lambda, (), ()(), ((()))()$ 。
 - 语言 $\{a^i (ab)^j (ca)^{2j} \mid i, j > 0\}$ 。
14. 利用泵引理说明下面的每个集合都不是正则的。
- $\{a, b\}$ 上的回文串的集合。
 - $\{a^n b^m \mid n < m\}$ 。
 - $\{a^i b^j c^{2j} \mid i \geq 0, j \geq 0\}$ 。
 - $\{ww \mid w \in \{a, b\}^*\}$ 。
- *e) 无限串的初始序列的集合
- $$abaabaaabaaaab \dots ba^n ba^{n+1} b \dots$$
- $\{a, b\}$ 上的串, 串中 a 的个数为完全立方。

218

15. 证明 $\{a, b\}$ 上的非回文的集合不是正则语言。
16. 设 L 是正则语言, $L_1 = \{uu \mid u \in L\}$ 。 L_1 一定是正则的吗? 证明你的答案。
17. 设 L_1 是非正则语言, L_2 是任意的有限语言。
- 证明 $L_1 \cup L_2$ 不是正则语言。
 - 证明 $L_1 - L_2$ 不是正则语言。
 - 说明当 L_2 是有限的时候, a) 和 b) 的结论不成立。
18. 给出 $\{a, b\}$ 上的满足下面描述的语言 L_1 和 L_2 的例子。
- L_1 是正则的, L_2 不是正则的, $L_1 \cup L_2$ 是正则的。
 - L_1 是正则的, L_2 不是正则的, $L_1 \cup L_2$ 不是正则的。
 - L_1 是正则的, L_2 不是正则的, $L_1 \cap L_2$ 是正则的。
 - L_1 不是正则的, L_2 不是正则的, $L_1 \cup L_2$ 是正则的。
 - L_1 不是正则的, 但是 L_1^* 是正则的。
- *19. 设 Σ_1 和 Σ_2 是两个字母表。串同态 (String Homomorphism) 是从 Σ_1^* 到 Σ_2^* 保持连接的一个全函数 h 。 h 满足下面的条件:
- $h(\lambda) = \lambda$ 。
 - $h(uv) = h(u)h(v)$ 。
- 设 $L_1 \subseteq \Sigma_1^*$ 是一个正则语言。说明集合 $\{h(w) \mid w \in L_1\}$ 在 Σ_2 上是正则的。这个集合叫做 L_1 在 h 下的同态象 (Homomorphic Image)。
 - 设 $L_2 \subseteq \Sigma_2^*$ 是一个正则语言。说明集合 $\{w \in \Sigma_1^* \mid h(w) \in L_2\}$ 是正则的。这个集合叫做 L_2 在 h 下的逆象 (Inverse Image)。
20. 上下文无关文法 $G = (V, \Sigma, P, S)$ 称为右线性 (Right-Linear) 的, 如果文法的每一条规则具有下面的形式:
- $A \rightarrow u$, 或者
 - $A \rightarrow uB$
- 其中, $A, B \in V, u \in \Sigma^*$ 。利用 6.3 节之中的方法说明右线性文法产生的恰好是正则集合。
- *21. 上下文无关文法 $G = (V, \Sigma, P, S)$ 称为左正则 (Left-Regular) 的, 如果文法的每一条规则具有下面的形式:
- $A \rightarrow \lambda$,
 - $A \rightarrow a$, 或者
 - $A \rightarrow Ba$
- 其中 $A, B \in V, a \in \Sigma$
- 设计一个算法来构造接收左正则文法的语言的正则 NFA
 - 说明左正则文法产生的正好是正则集。
22. 上下文无关文法 $G = (V, \Sigma, P, S)$ 称为左线性 (Left-Linear) 的, 如果文法的每一条规则具有下面的形式:
- $A \rightarrow u$, 或者
 - $A \rightarrow Bu$
- 其中, $A, B \in V, u \in \Sigma^*$ 。说明左线性文法产生的正好是正则集。
23. 给出一个语言 L 使得 \equiv_L 只有三个等价类。
24. 给出语言 a^*b^* 的 \equiv_L 等价类。
25. 设 $[u]_{\equiv_L}$ 是语言 L 的一个 \equiv_L 等价类, 说明如果 $[u]_{\equiv_L}$ 包含一个串 $v \in L$, 那么 $[u]_{\equiv_L}$ 中所有的串都在 L 中。
26. 证明对于所有的正则语言 L , \equiv_L 是右恒定的。即是, 如果 $u \equiv_L v$, 那么对任意的 $x \in \Sigma^*$, 有 $ux \equiv_L vx$, 其中 Σ 是 L 的字母表。
27. 利用 Myhill-Nerode 定理证明语言 $\{a^i \mid i \text{ 是完全平方数}\}$ 不是正则的。

28. 设 $u \in [ab]_{\equiv_M}$ 和 $v \in [aba]_{\equiv_M}$ 是在例 6.7.4 中 $(a \cup b)(a \cup b^*)$ 的等价类中的串。说明 u 和 v 是不可区分的。
29. 给出例 5.3.1 中 DFA 的由关系 \equiv_M 定义的等价类。
30. 给出例 5.3.3 中 DFA 的由关系 \equiv_M 定义的等价类。
31. 设 M_L 是在定理 6.7.4 和定理 6.7.5 中定义的接收语言 L 的最小状态的 DFA，设 M 是和 M_L 具有同样数目的接收 L 的 DFA。证明 M_L 和 M 是一样的除了（可能）它们中的状态图的名字不一样。这样的两个 DFA 被称作是同构的（Isomorphic）。

参考文献注释

Kleene [1956] 中提出了正则集合和有限自动机接收的语言是等价的。在 6.2 节中给出的证明利用了 McNaughton 和 Yamada [1960] 中的模型。Chomsky 和 Miller [1958] 确定了正则文法产生的语言和有限自动机接收的语言的等价性。在同态下的封闭性（练习 19）来自 Ginsburg 和 Rose [1963b]。正则集合逆的封闭性由 Rabin 和 Scott [1959] 提出。此外，在 Bar-Hillel、Perles 和 Shamir [1961] 以及 Ginsburg [1966] 和 Rose [1963b] 中可以找到正则集合结果的封闭性。正则集合的泵引理由 Bar-Hillel、Perles 和 Shamir [1961] 提出。语言的等价类的数目和正则性之间的关系在 Myhill [1957] 和 Neorode [1958] 中得到了确立。

第7章 下推自动机和上下文无关语言

正则语言被认为是由正则文法产生的语言和被有限自动机接收的语言。本章将给出一类自动机,叫做下推自动机,它接收上下文无关语言。下推自动机是增加了额外的栈内存的有限状态机。额外的栈提供给下推自动机后进先出的内存管理能力。栈和状态的结合克服了内存限制,这种限制使确定型有限自动机无法接收语言 $\{a^ib^i \mid i \geq 0\}$ 。

和正则语言一样,上下文无关语言的泵引理确保了上下文无关语言串中的重复子串的存在。泵引理提供了一种技术来说明许多很容易定义的语言不是上下文无关的。

7.1 下推自动机

定理 6.5.1 中确定了语言 $\{a^ib^i \mid i \geq 0\}$ 是不能够被任何一个有限自动机接收的。为了接收该语言,自动机需要具有记录处理任意有限个 a 的过程的能力。自动机只能够含有有限个状态的约束,使得它不能“记录”在一个任意的输入串的前面已经处理过的 a 的个数。通过对有限自动机增加根据状态、无限的存储进行转换的能力,从而使其具有使用无限内存的能力,这样我们就构造出了一种新的自动机。

221. 添加一个下推栈或者只是简单地添加一个栈到有限自动机中,就构成了一个新的机器,即下推自动机(PDA)。栈操作只影响栈顶的元素,出栈将栈顶的元素从栈中删除,而进栈则将一个元素放到栈顶。定义 7.1.1 对下推自动机的概念进行了形式化。PDA 的各个组成部分 Q 、 Σ 、 q_0 和 F 与有限自动机中的相同。

定义 7.1.1 一个下推自动机(pushdown automaton)是一个六元组 $(Q, \Sigma, \Gamma, \delta, q_0, F)$,其中 Q 是状态的有限集, Σ 是输入字母的有限集, Γ 是栈字母表的有限集合, q_0 是开始状态, $F \subseteq Q$ 是终止状态的集合, δ 是从 $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$ 到 $Q \times (\Gamma \cup \{\lambda\})$ 的子集的转换函数。

PDA 具有两个字母表:一个是输入串使用的输入字母表 Σ ,一个是栈字母表 Γ ,它的元素存放在栈中。栈表示为栈元素的一个串,栈顶的元素是串中的最左字符。我们将使用大写字母来表示栈元素,希腊字母表示栈中的串。记号 $A\alpha$ 表示栈顶元素为 A 的栈。空栈记为 λ 。PDA 的计算开始时,机器处于状态 q_0 ,输入在磁带上,并且栈为空。

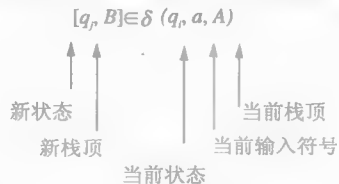
PDA 考虑当前的状态、输入符号和栈顶符号来决定机器的转换。转换函数 δ 列举了给定状态、符号和栈顶元素的所有可能的转换。

$$\delta(q_i, a, A) = \{[q_j, B], [q_k, C]\}$$

上面的转换函数的值表明当自动机在状态 q_i 扫描到符号 a 并且栈顶为 A 时具有两个可能的转换,转换会导致机器

- i) 状态从 q_i 改变到 q_j ,
- ii) 处理符号 a (带头前移)
- iii) 将 A 从栈顶删除(弹出栈),和
- iv) 将 B 压入栈顶。

因为可能为一个机器格局指定了多个转化,因此 PDA 是非确定型自动机。



222. 下推自动机也可以用状态图来描述。弧上的标记表明输入和栈操作。转换 $\delta(q_i, a, A) = \{[q_j, B]\}$ 描述为:



符号/表示替换： A/B 表明 A 替换栈顶的符号 B 。

转换函数的定义域为 $Q \times (\Sigma \cup \lambda) \times (\Gamma \cup \lambda)$ ，它说明 λ 可以出现在一个转换的输入或者栈顶的位置。一个 λ 指定了组件上的值既不会被考虑也不对转换起作用，转换的应用由非 λ 的位置完全决定。

当 λ 作为一个在转换函数的栈位置的参数出现时，只要当前状态和输入符号与转换中的相匹配，就可以应用转换，而不用考虑栈中的状态。栈顶可以包含任何的符号或者栈为空。只要机器处于状态 q_i 并扫描到 a ，就可以应用转换 $[q_i, B] \in \delta(q_i, a, \lambda)$ 。转换的应用会导致机器进入状态 q_j ，并将符号 B 添加到栈顶。

符号 λ 可能出现在一个转换的新的栈位置处， $[q_i, \lambda] \in \delta(q_i, a, A)$ 。在执行这样的一个转换时，将不会把任何符号压入栈中。我们将会看几个例子，从而了解 PDA 转换中 λ 会产生怎样的影响。

如果输入位置是 λ ，那么转换不处理输入字符。所以转换 (i) 出栈和 (ii) 将栈符号 A 压入栈中，而不改变状态和输入。

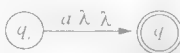
i) $[q_i, \lambda] \in \delta(q_i, \lambda, A)$



ii) $[q_i, A] \in \delta(q_i, \lambda, \lambda)$



iii) $[q_i, \lambda] \in \delta(q_i, a, \lambda)$



如果转换指定的动作在新的栈顶位置是 λ ， $[q_i, \lambda]$ ，那么将不会把任何符号压入到栈中。转换 (iii) 中的 PDA 等价于一个有限状态自动机。是否可以应用转换是由状态和输入符号决定的，转换并不会考虑或者更改栈。 [223]

PDA 的格局由三元组 $[q_i, w, \alpha]$ 来描述，其中 q_i 是机器状态， w 是未处理的输入， α 是栈记号。

$$[q_i, w, \alpha] \vdash [q_j, v, \beta]$$

表明格局 $[q_j, v, \beta]$ 可以从 $[q_i, w, \alpha]$ 通过 DFA M 的一次转换获得。和前面的一样， \vdash 描述了一系列转换的结果。当不存在歧义的时候，可省略写在下方的 M。PDA 的计算是一系列从机器的初始状态和空栈开始的转换。

我们现在要构造一个接收语言 $\{a^i b^i \mid i \geq 0\}$ 的 PDA M。在计算开始时，输入串为 w ，栈为空。处理输入符号 a 导致 A 被压入栈中。处理 b 出栈，则匹配 b 和 a 的个数。下图展示了输入串为 $aabb$ 时，M 的计算过程。

$M: Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

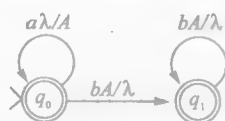
$\Gamma = \{A\}$

$F = \{q_0, q_1\}$

$\delta(q_0, a, \lambda) = \{[q_0, A]\}$

$\delta(q_0, b, A) = \{[q_1, \lambda]\}$

$\delta(q_1, b, A) = \{[q_1, \lambda]\}$



$[q_0, aabb, \lambda]$

$\vdash [q_0, abb, A]$

$\vdash [q_0, bb, AA]$

$\vdash [q_1, b, A]$

$\vdash [q_1, \lambda, \lambda]$

输入 $a^i b^i$ ，M 处理整个串，并停止在接收状态，此时栈为空。这些条件是 PDA 的接收准则。

定义 7.1.2 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是 PDA。如果存在一个计算

$$[q_0, w, \lambda] \vdash [q_i, \lambda, \lambda]$$

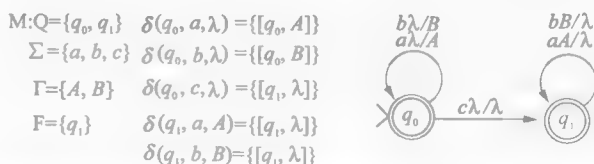
其中, $q_i \in F$, 则串 $w \in \Sigma^*$ 被 M 接收, M 的语言记做 $L(M)$, 它是 M 接收的字符串的集合。

一个接收串的计算称作是成功的。处理整个输入并终止在非接收状态的计算称作是不成功的。因为转换函数的非确定性, 某些计算可能不能完全处理完输入串。这种形式的计算也被认为是不成功的。

224

PDA 的接收遵循非确定型自动机的标准模式。处理完整个串并停止在接收状态的一个计算足以说明串在该 PDA 的语言中。存在其他的不成功的计算并不影响串的接收。

例 7.1.1 PDA M 接收语言 $\{wcw^R \mid w \in \{a, b\}^*\}$ 。采用栈来记录处理过的 w 。栈符号 A 和 B 分别表示输入的是 a 和 b 。



一个成功的计算在处理到 w 的时候会于栈中记录 w 。一旦遇到 c 则进入接收状态 q_1 , 栈中包含的串表示为 w^R 。用余下的输入和栈中的元素匹配来完成计算。 M 处理输入 $abcba$ 的计算为

$$\begin{aligned}
 & [q_0, abcba, \lambda] \\
 & \vdash [q_0, bcba, A] \\
 & \vdash [q_0, cba, BA] \\
 & \vdash [q_1, ba, BA] \\
 & \vdash [q_1, a, A] \\
 & \vdash [q_1, \lambda, \lambda]
 \end{aligned}$$

□

PDA 是确定的, 如果对于每一个状态、输入符号和栈顶的组合, 最多存在一个可以应用的转换。两个转换 $[q_i, C] \in \delta(q_i, u, A)$ 和 $[q_i, D] \in \delta(q_i, v, B)$ 称作是兼容的 (Compatible), 如果满足下面的任何一个条件:

- i) $u = v$ 并且 $A = B$
- ii) $u = v$ 并且 $A = \lambda$ 或者 $B = \lambda$
- iii) $A = B$ 并且 $u = \lambda$ 或者 $v = \lambda$
- iv) $u = \lambda$ 或者 $v = \lambda$ 并且 $A = \lambda$ 或者 $B = \lambda$

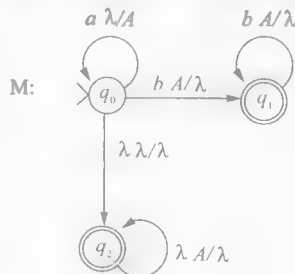
兼容转移可以应用到相同的机器格局中。如果一个 PDA 不包含不同的, 那么它就是确定的。例 7.1.1 中的 PDA 和构造的接收 $\{a^i b^i \mid i \geq 0\}$ 的自动机都是确定的。

225

例 7.1.2 语言 $L = \{a^i \mid i \geq 0\} \cup \{a^i b^i \mid i \geq 0\}$ 包含完全由 a 组成的串或者具有相同数目的 a 和 b 的串。接收 L 的 PDA M 的栈会保留一个已处理的 a 的个数的记录, 直到遇到一个 b 或者输入串被处理完成。

当在状态 q_0 扫描到 a 时, 存在两个相容的转换。具有形式 $a^i b^i$ ($i > 0$) 的串, 被一个处在状态 q_0 和 q_1 的计算接收。如果进入 q_2 的转换跟随着处理 a^i 中的最后的一个 a , 则栈被清空, 并且接收输入。以任何其他的方式到达状态 q_2 , 将导致计算不成功, 因为在状态 q_2 之后不会处理输入。

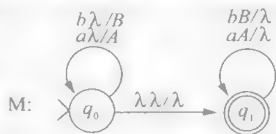
λ -转换允许 M 任何时候进入 q_2 , 只要它处于状态 q_0 。这个转换将非确定性引入到 M 的计算中。接收串 a^i 的计算在状态 q_0 处理整个串, 转换到 q_2 , 清空栈, 并接收串。



□

例 7.1.3 $\{a, b\}$ 上的长度为偶数的回文串可以被下面的 PDA 接收

即, $L(M) = \{ww^R \mid w \in \{a,b\}^*\}$ 。一个成功的计算在处理串 w 时一直处于状态 q_0 , 并且一旦读到 w^R 的第一个符号就会进入状态 q_1 。与例 7.1.1 中的串不同, L 中的串并不包含一个中间的标记, 该标识使得状态从 q_0 转换到 q_1 。非确定型允许机器猜测何时到达串的中间。到状态 q_1 的转换, 如果不是在处理完 w 的最后一个元素时立即出现, 则将导致计算不成功。□



在第5章中, 我们展示了确定型和非确定型自动机接收同一个语言族。

非确定型在设计功能的时候是有用的, 而它并不增加自动机接收串的能力。但是这对下推自动机却不成立。

226

从例 7.1.3 中我们知道, 不存在接收语言 $L = \{ww^R \mid w \in \{a,b\}^*\}$ 的确定型下推自动机。我们直观地考虑 PDA 接收语言 L 所需要的性质。因为 PDA 的计算以从左到右的形式处理输入, 所以自动机并不能够确定什么时候前半部分输入已经读完。而在例 7.1.3 中的非确定型自动机就不会引起这种问题。从 q_0 到 q_1 的转换表示一个非确定型的猜测: 正在扫描的符号是第二个 w 中的第一个符号。对一个 L 中的串, 总有一个猜测是正确的, 这使得计算能够匹配后半部串和栈中的元素, 并接收输入。

考虑一个确定型 PDA 处理下面输入串时的可能动作:

aabbbaa 和 *aabbbbbaa*

当读到串的前半部分的 a 或者 b 时, 相应的栈符号 A 或者 B 被压入到栈中, 这将被用于比较后半部分的输入。读完前三个字符, 栈中的符号为 BAA 。不管处理的是哪个串, 下一个输入符号都是 b 。为了接收 *aabbbaa*, 必须出栈, 以便开始用 *aab* 匹配 *baa*。然而, 如果接收的是 *aabbbbbaa*, 那么自动机必须把 B 压入栈中。确定型机器对这种配置只有一种选择, 因此两个串中必有一个不被接收。

确定型下推自动机接收的语言包括所有的正则语言, 以及上下文无关语言的一个适当的子集。这个语言族对于编程语言的设计是非常重要的, 它由 $LR(k)$ 文法产生的语言组成。用 $LR(k)$ 文法定义语言和确定性分析将在 19 章中介绍。

7.2 PDA 的变种

下推自动机经常使用与定义 7.1.1 稍有不同的形式进行定义。在本节中, 我们将介绍几种不同的定义, 它能保存接收的语言。

除改变状态以外, PDA 的转换还包括其他的三个动作: 出栈, 将一个元素入栈以及处理输入符号。PDA 称为是原子的 (atomic), 如果每一个转换只引发这三个动作中的一个。原子 PDA 具有下面的形式

- i) $[q_i, \lambda] \in \delta(q_i, a, \lambda)$
- ii) $[q_j, \lambda] \in \delta(q_i, \lambda, A)$ 或者
- iii) $[q_j, A] \in \delta(q_i, \lambda, \lambda)$ 。

227

很显然, 每一个原子 PDA 都是定义 7.1.1 中的一个 PDA。定理 7.2.1 说明了原子 PDA 接收的语言和其他的 PDA 接收的语言是一样的。此外, 它给出了一个从任意的 PDA 构造等价的原子 PDA 的方法。

定理 7.2.1 设 M 是一个 PDA。则存在一个原子的 PDA M' , 使得 $L(M') = L(M)$ 。

证明: 为了构造 M' , 我们用一系列的原子转换替换 M 中的非原子转换。设 $[q_j, B] \in \delta(q_i, a, A)$ 是 M 的一个转换。原子等价需要两个新的状态 p_1 和 p_2 , 以及下面的转换

$$\begin{aligned} [p_1, \lambda] &\in \delta(q_i, a, \lambda) \\ \delta(p_1, \lambda, A) &= \{[p_2, \lambda]\} \\ \delta(p_2, \lambda, \lambda) &= \{[q_j, B]\} \end{aligned}$$

来实现与非原子单步转换相同的结果。

用类似的方式, 一个包括改变状态以及执行两个额外的动作的转换可以用两个原子转换的序列来替换。用一系列的原子转换替换所有的非原子转换将获得一个等价的原子 PDA。■

扩展 PDA 的转换, 使得 PDA 每次压入一串元素到栈中, 而不是单个元素。转换 $[q_j, BCD] \in$

$\delta(q_i, a, A)$ 将 BCD 压入到栈中, 从而 B 成为了新的栈顶。包含扩展转换的 PDA 称为扩展的 PDA。这种明显的一般化并没有增加下推自动机接收语言的能力。每一个扩展的 PDA 都可以转换成与定义 7.1.1 中的形式等价的 PDA。

为了从一个扩展的 PDA 构造一个 PDA, 需要将扩展转换转化为一系列简单的转换, 这些转换每次只将一个栈符号压入栈中。一次将 k 个元素压入栈中的扩展转换, 需要 $k-1$ 个额外的状态才能够达到。转换的序列

$$\begin{aligned}[p_1, D] &\in \delta(q_i, a, A) \\ \delta(p_1, \lambda, \lambda) &= \{[p_2, C]\} \\ \delta(p_2, \lambda, \lambda) &= \{[q_j, B]\}\end{aligned}$$

将串 BCD 压入栈中, 并使自动机处于 q_i 状态。连续地执行这三个转换将获得与执行单一的扩展转换 $[q_j, BCD] \in \delta(q_i, a, A)$ 一样的结果。前面的讨论可以概括为定理 7.2.2。

228 定理 7.2.2 设 M 是扩展的 PDA, 则存在 PDA M' , 使得 $L(M') = L(M)$ 。

例 7.2.1 设 $L = \{a^i b^{2i} \mid i \geq 1\}$, 分别构造接收 L 的标准 PDA, 原子 PDA 和扩展的 PDA。每个自动机具有相同的输入字母表 $\{a, b\}$ 、栈字母表 $\{A\}$ 和接收状态 q_1 。状态和转换如下:

PDA	原子 PDA	扩展的 PDA
$Q = \{q_0, q_1, q_2\}$	$Q = \{q_0, q_1, q_2, q_3, q_4\}$	$Q = \{q_0, q_1\}$
$\delta(q_0, a, \lambda) = \{[q_2, A]\}$	$\delta(q_0, a, \lambda) = \{[q_3, \lambda]\}$	$\delta(q_0, a, \lambda) = \{[q_0, AA]\}$
$\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$	$\delta(q_3, \lambda, \lambda) = \{[q_2, A]\}$	$\delta(q_0, b, A) = \{[q_1, \lambda]\}$
$\delta(q_0, b, A) = \{[q_1, \lambda]\}$	$\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$	$\delta(q_1, b, A) = \{[q_1, \lambda]\}$
$\delta(q_1, b, A) = \{[q_1, \lambda]\}$	$\delta(q_0, b, \lambda) = \{[q_4, \lambda]\}$	
	$\delta(q_4, \lambda, A) = \{[q_1, \lambda]\}$	
	$\delta(q_1, b, \lambda) = \{[q_4, \lambda]\}$	

正如所期望的那样, 原子 PDA 比标准 PDA 需要更多的转换, 而扩展的 PDA 则需要较少的转换。栈符号 A 用来记录匹配的 b 的个数。扩展转换 $\delta(q_0, a, \lambda) = \{[q_0, AA]\}$ 一次将两个计数压入栈中。为了达到同样的结果, 标准的 PDA 需要两步转换而原子 PDA 需要三步转换。□

根据定义 7.1.2, 如果存在一个计算, 它处理完整个串并停止在接收状态而且栈为空, 那么串被接收。这种接收称为被终结状态空栈接收。单独以终结状态或者栈格局的形式定义接收并不改变下推自动机识别的语言的集合。

如果存在计算 $[q_0, w, \lambda] \vdash^* [q_f, \lambda, \alpha]$, 那么串 w 被终结状态接收, 其中 q_f 是接收状态, $\alpha \in \Gamma^*$ 。即, 存在一个处理完串、并终止在接收状态的计算。终止时的栈内容 α 与是否被终结状态接收是不相关的。被终结状态接收的语言记做 L_F 。

引理 7.2.3 L 是被 PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 终结状态接收的语言。那么, 存在一个被终结状态空栈接收 L 的 PDA。

证明: PDA $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, \{q_f\})$ 是通过在 M 中增加状态 q_f 和与 q_f 相关的转换获得的。直观地, M' 的接收串的计算应该和 M 中的一样, 除了额外的清空栈的转换。转换函数 δ' 通过在 δ 中增加下列转换而获得:

$$\begin{aligned}\delta'(q_i, \lambda, \lambda) &= \{[q_f, \lambda]\} \text{ 对所有的 } q_i \in F \\ \delta'(q_f, \lambda, A) &= \{[q_f, \lambda]\} \text{ 对所有的 } A \in \Gamma\end{aligned}$$

229 设 $[q_0, w, \lambda] \vdash^* [q_f, \lambda, \alpha]$ 是 M 中终结状态接收 w 的计算。在 M' 中, 通过进入接收状态 q_f 并清空栈来完成该计算

$$\begin{aligned}[q_0, w, \lambda] \\ \vdash^* [q_i, \lambda, \alpha] \\ \vdash^* [q_f, \lambda, \alpha] \\ \vdash^* [q_f, \lambda, \lambda]\end{aligned}$$

展示了 M' 中如何接收 w 。

我们必须保证新的转换不会导致 M' 接收不在 $L(M)$ 中的串。 M' 中唯一的接收状态是 q_f ，只有当一个转换是从 M 的任意接收状态开始时才能够进入 q_f 。因为转移到 q_f 的转换并不处理输入，所以如果进入 q_f 时有未处理的输入就会导致计算失败。因此，串 w 被 M' 接收当且仅当 M 中存在一个计算，该计算处理完整个 w 并终止在 M 的接收状态。这就是说， $w \in L(M')$ 当且仅当 $w \in L(M)$ 。■

如果存在计算 $[q_0, w, \lambda] \vdash^* q_f, \lambda, \lambda$ ，则称 w 被空栈接收。对于计算终止的状态 q_f 没有限制。当定义一个空栈接收时，有必要要求至少存在一个转换来允许接收不包含空串语言。空栈接收的语言记为 $L_E(M)$ 。

引理 7.2.4 设 L 是 PDA $M = (Q, \Sigma, \Gamma, \delta, q_0)$ 空栈接收的语言。那么，存在终结状态空栈接收 L 的 PDA M' 。

证明： 设 $M' = (Q \cup \{q_0'\}, \Sigma, \Gamma, \delta', q_0', Q)$ ，其中对每一个 $q_i \in Q, x \in \Sigma \cup \{\lambda\}$ 和 $A \in \Gamma \cup \{\lambda\}$ 都有 $\delta'(q_i, x, A) = \delta(q_i, x, A), \delta'(q_0', x, A) = \delta(q_0, \lambda, A)$ 。原始自动机的每一个状态都是 M' 的一个接收状态。

M' 和 M 的计算是一样的，除了 M 和 M' 中分别以状态 q_0 和 q_0' 开始外。 M' 中长度大于 1 的计算终止时栈为空，同时也终止在接收状态。因为 q_0 不是接收状态，所以 M' 接收空串当且仅当它被 M 接收。所以 $L(M') = L_E(M)$ 。

引理 7.2.3 和引理 7.2.4 说明了终结状态或者空栈接收的语言都可以被终结状态空栈接收。练习 8 和练习 9 说明了终结状态空栈接收的语言同时也可以被具有较少约束形式的下推自动机接收。这些观察得到了下面的定理。

定理 7.2.5 下面三个条件是等价的：

- i) L 被某个 PDA 接收。
- ii) 存在 PDA $M_1, L_E(M_1) = L$ 。
- iii) 存在 PDA $M_2, L_E(M_2) = L$ 。

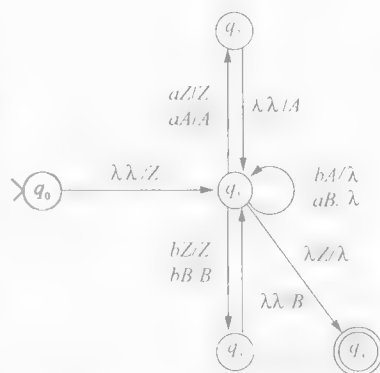
我们已经见到了标准 PDA 模型的几种变形，这主要是通过改变接收准则和转换形式实现的。另一种普遍的修改是假设存在用于标记栈底的可区分的元素。这种修改可以读取栈底标记，但是不能出栈。通过读取栈底符号来允许机器识别空栈并采取相应的动作。下面的例子展示了栈底标记并说明了在标准的 PDA 中如何模拟。

例 7.2.2 下推自动机 M 由下面的转换定义

它接收具有相同数目的 a 和 b 的串。栈符号 Z 扮演栈底标记的角色；在第一个转换中将它放到栈中并在整个计算中一直维持在栈底。

栈记录已经读过的 a 和 b 的数目的差。如果自动机处理的 a 比 b 多 n 个，那么栈中将包含 n 个 A ；类似地，栈中包含 n 个 B 表明处理过的 b 比 a 多 n 个。当读到栈底标记 Z 时则表明处理到的 a 和 b 的数目相等。计算

$[q_0, abba, \lambda]$
 $\vdash [q_1, abba, Z]$
 $\vdash [q_2, bba, Z]$
 $\vdash [q_1, bba, AZ]$
 $\vdash [q_1, ba, Z]$
 $\vdash [q_3, a, Z]$
 $\vdash [q_1, a, BZ]$



[231]

$$\vdash [q_1, \lambda, Z]$$

$$\vdash [q_1, \lambda, \lambda]$$

展示了接收 $abba$ 。当读到 a 并且栈顶是 A 或者 Z 时, 通过进入状态 q_1 将一个 A 添加到栈中, 并返回到 q_1 。如果栈顶是 B , 则在状态 q_1 出栈, 因为读到的 a 减少了处理到的 a 和 b 数目的差。当读到 b 时, 应用类似的策略。

自动机唯一的接收状态是 q_1 。如果输入串中具有相同数目的 a 和 b , 转移到 q_1 的转换将会弹出 Z , 并终止计算。□

接收相同语言的下推自动机的各种变形展示了采用栈存储接收的健壮性。在下一节中, 我们将展示下推自动机接收的语言正好是上下文无关语言。

7.3 上下文无关语言的接收

在第6章中, 我们介绍了正则文法产生的语言和 DFA 接收的语言一样。本节将继续讨论文法产生的语言和自动机接收的语言的关系。下推自动机作为上下文无关语言的接收器, 它的特征是通过确定 PDA 的计算和上下文无关文法的推导之间的关系获得的。

首先我们将证明, 每一个上下文无关的语言可以被一个扩展的 PDA 接收。为了完成证明, 我们采用文法的规则来产生等价的 PDA 的转换。设 L 是上下文无关的语言, G 是满足格立巴赫范式的文法, 并且 $L(G) = L$ 。 G 的产生式规则, 除 $S \rightarrow \lambda$ 之外, 具有如下形式: $A \rightarrow aA_1A_2 \cdots A_n$ 。在最左推导之中, 必须以从左到右的顺序处理变量 A_1 。将变量 $A_1A_2 \cdots A_n$ 按推导所需要的顺序压入栈中。PDA 具有两个状态: 开始状态 q_0 和接收状态 q_1 。形式为 $S \rightarrow aA_1A_2 \cdots A_n$ 的推导规则 S 产生一个转换, 这个转换处理终结符 a , 将 $A_1A_2 \cdots A_n$ 压入栈中, 并进入状态 q_1 。计算的剩下部分是利用输入符号和栈顶来决定适当的转换。

下面用接收语言 $\{a^i b^i \mid i > 0\}$ 的格立巴赫范式文法 G 来展示如何构造等价的 PDA。

$$G: S \rightarrow aAB \mid aB$$

$$A \rightarrow aAB \mid aB$$

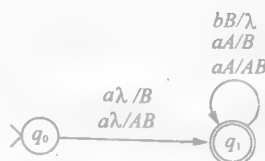
$$B \rightarrow b$$

等价 PDA 的转换函数直接从 G 的产生式规则定义:

$$\delta(q_0, a\lambda) = \{[q_1, AB], [q_1, B]\}$$

$$\delta(q_1, a, A) = \{[q_1, AB], [q_1, B]\}$$

$$\delta(q_1, b, B) = \{[q_1, \lambda]\}$$



[232]

下面处理串 $aaabbb$ 的计算展示了格立巴赫范式文法的推导和相应的 PDA 的计算之间的关系。

$$\begin{array}{ll} S \Rightarrow aAB & [q_0, aaabbb, \lambda] \vdash [q_1, aaabbb, AB] \\ \Rightarrow aaABB & \vdash [q_1, abbb, ABB] \\ \Rightarrow aaaBBB & \vdash [q_1, bbb, BBB] \\ \Rightarrow aaabBB & \vdash [q_1, bb, BB] \\ \Rightarrow aaabbB & \vdash [q_1, b, B] \\ \Rightarrow aaabbb & \vdash [q_1, \lambda, \lambda] \end{array}$$

推导产生一个前缀终结符后紧跟着后缀变量的串。处理输入符号对应着推导之中的该变量的产生。PDA 的栈中包含导出串中的变量。定理 7.3.1 形式化了这种从格立巴赫范式文法产生等价的 PDA 的策略。它说明了每个上下文无关的文法都能够被一个 PDA 接受。

定理 7.3.1 设 L 是一个上下文无关的语言。那么存在一个接收 L 的 PDA。

证明: 设 $G = (V, \Sigma, P, S)$ 是产生语言 L 的格立巴赫范式文法。开始状态为 q_0 的, 接收 L 的扩展

的 PDA M 的定义如下:

$$Q_M = \{q_0, q_1\}$$

$$\Sigma_M = \Sigma$$

$$\Gamma_M = V - \{S\}$$

$$F_M = \{q_1\}$$

它的转换

$$\delta(q_0, a, \lambda) = \{[q_1, w] \mid S \rightarrow aw \in P\}$$

$$\delta(q_1, a, A) = \{[q_1, w] \mid A \rightarrow aw \in P \text{ 和 } A \in V - \{S\}\}$$

$$\delta(q_0, \lambda, \lambda) = \{[q_1, \lambda]\} \text{ 如果 } S \rightarrow \lambda \in P$$

接收 L 。

首先证明 $L \subseteq L(M)$ 设 $S \Rightarrow^* uw$ 是一个推导, 其中 $u \in \Sigma^+, w \in V^+$ 下面将证明 M 中存在计算

$$[q_0, u, \lambda] \vdash [q_1, \lambda, w]$$

对推导长度进行归纳并利用 G 中的推导和 M 的计算之间的关系进行证明

[233]

归纳的基础是含有长度为 1 的推导 $S \Rightarrow aw$ 规则 $S \rightarrow aw$ 产生的转换能够得到所要求的计算。假设由推导 $S \Rightarrow^* uw$ 产生的所有的串 uw 在 M 中存在计算:

$$[q_0, u, \lambda] \vdash [q_1, \lambda, w]$$

现在, 设 $S \xRightarrow{n+1} uw$ 是一个推导, 其中 $u = va \in \Sigma^+, w \in V^+$ 。推导可以写成:

$$S \xRightarrow{n} vAw_2 \Rightarrow uw$$

其中, $w = w_1w_2$ 且 $A \rightarrow aw_1$ 是 P 中的规则。归纳假设和转换 $[q_1, w_1] \in \delta(q_1, a, A)$ 相结合, 得到计算

$$[q_0, va, \lambda] \vdash [q_1, a, Aw_2]$$

$$\vdash [q_1, \lambda, w_1w_2]$$

对 L 中的每一个具有整数长度的串 u , M 中对应于推导 $S \Rightarrow^* u$ 的计算接收 u 。如果 $\lambda \in L$, 那么 $S \rightarrow \lambda$ 是 G 中的一个规则, 并且计算 $[q_0, \lambda, \lambda] \vdash [q_1, \lambda, \lambda]$ 接收空串。

相反, $L(M) \subseteq L$, 即是证明对于每一个计算 $[q_0, u, \lambda] \vdash [q_1, \lambda, w]$, G 中存在与之对应的推导 $S \Rightarrow^* uw$ 。证明通过对计算中的转换的数目归纳, 将留做练习。

为了完成描述上下文无关的语言正好是下推自动机接收的语言, 我们必须说明 PDA 接收的每一个语言都是上下文无关的语言。因为可以通过自动机的转换构造上下文无关文法的规则, 所以规则的运用和 PDA 的计算中的转换相对应。为了简化证明, 我们将这个过程分成四个阶段:

1. 在 PDA 中加入的转换, 使得语言中的每一个串都能够被计算接收, 并且该计算中的每一个转换都包含出栈和入栈操作。

2. 从修改的 PDA 中构造文法的规则。

3. 给出一个展示 PDA 的计算和文法推导之间的关系的例子。

4. 最后, 形式证明文法和 PDA 的语言是相同的。

前两个阶段是构造性的。添加转换和构造文法规则。最后一步由引理 7.3.3 和引理 7.3.4 完成, 它说明了规则产生的串正好是 PDA 接收的串。首先从任意一个 PDA M 开始, 说明 $L(M)$ 是上下文无关的。证明首先修改 M , 使得转换能够转化成规则。

设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是 PDA。扩展自动机 PDA M' 的转换函数为 δ' , 它在 M 的转换函数 δ 上增加了如下的转换:

i) 对每一个 $A \in \Gamma$, 如果 $[q_j, \lambda] \in \delta(q_i, u, \lambda)$, 那么 $[q_j, A] \in \delta'(q_i, u, A)$,

ii) 对每一个 $A \in \Gamma$, 如果 $[q_j, B] \in \delta(q_i, u, \lambda)$, 那么 $[q_j, BA] \in \delta'(q_i, u, A)$ 。

这些转换的解释是: M 的并不从栈中删出一个元素的转换, 可以看成是一个初始的出栈操作和之后的用同一个符号替换栈顶的操作。一个利用新的转换接收的串的计算也可以通过原始的转换获得; 所以, $L(M) = L(M')$ 。

文法 $G = (V, \Sigma, P, S)$ 从 M' 的转换中构造而来。 G 的字母表是 M' 的输入字母表。 G 中的变量包含开始符号 S 和形式为 $\langle q_i, A, q_j \rangle$ 的对象, 其中 q 是 M' 的状态, $A \in \Gamma \cup \{\lambda\}$ 。 变量 $\langle q_i, A, q_j \rangle$ 表示一个从状态 q_i 开始, 以 q_j 结束的计算, 并把符号 A 从栈中删除。 G 的规则定义如下:

1. $S \rightarrow \langle q_0, \lambda, q_1 \rangle$, 对每一个 $q_1 \in F$
2. 每个转换 $[q_j, B] \in \delta'(q_i, x, A)$, 其中 $A \in \Gamma \cup \{\lambda\}$, 产生下面规则的集合:

$$\{\langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_k \rangle \mid q_k \in Q\}$$
3. 每个转换 $[q_j, BA] \in \delta'(q_i, x, A)$, 其中 $A \in \Gamma$, 产生下面规则的集合:

$$\{\langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_n \rangle \langle q_n, A, q_k \rangle \mid q_k, q_n \in Q\}$$
4. 对每一个状态 $q_k \in Q$, 有 $\langle q_k, \lambda, q_k \rangle \rightarrow \lambda$ 。

推导从第1类规则开始。这类产生式的右边表示一个从 q_0 开始, 并终止在接收状态的计算, 该计算终止时栈为空。换句话说, 即是 M' 中的一个成功的计算。第2类和第3类规则跟踪机器的动作。第3类规则与 M' 的扩展转换相对应。在一个计算中, 这些规则增加栈的大小。相应的规则的作用是在推导之中引入额外的变量。

第4类规则用来终止推导。规则 $\langle q_k, \lambda, q_k \rangle \rightarrow \lambda$ 表示一个从状态 q_k 开始到其自身的计算, 这个计算不改变栈, 即它是一个空计算。

例 7.3.1 文法 G 是从 PDA M 构造出来的, M 的语言是集合 $\{a^n cb^n \mid n \geq 0\}$

$$\begin{aligned} M: Q &= \{q_0, q_1\} & \delta(q_0, a, \lambda) &= \{[q_0, A]\} \\ \Sigma &= \{a, b, c\} & \delta(q_0, c, \lambda) &= \{[q_1, \lambda]\} \\ \Gamma &= \{A\} & \delta(q_1, b, A) &= \{[q_1, \lambda]\} \\ F &= \{q_1\} \end{aligned}$$

在 M 中增加转换 $\delta'(q_0, a, A) = \{[q_0, AA]\}$ 和 $\delta'(q_0, c, A) = \{[q_1, A]\}$ 来构造 M' 。等价文法 G 的规则和从这些文法构造而来的转换如下所示

转 换	规 则
	$S \rightarrow \langle q_0, \lambda, q_1 \rangle$
$\delta(q_0, a, \lambda) = \{[q_0, A]\}$	$\langle q_0, \lambda, q_0 \rangle \rightarrow a \langle q_0, A, q_0 \rangle$ $\langle q_0, \lambda, q_1 \rangle \rightarrow a \langle q_0, A, q_1 \rangle$
$\delta(q_0, a, A) = \{[q_0, AA]\}$	$\langle q_0, A, q_0 \rangle \rightarrow a \langle q_0, A, q_0 \rangle \langle q_0, A, q_0 \rangle$ $\langle q_0, A, q_1 \rangle \rightarrow a \langle q_0, A, q_0 \rangle \langle q_0, A, q_1 \rangle$ $\langle q_0, A, q_0 \rangle \rightarrow a \langle q_0, A, q_1 \rangle \langle q_1, A, q_0 \rangle$ $\langle q_0, A, q_1 \rangle \rightarrow a \langle q_0, A, q_1 \rangle \langle q_1, A, q_1 \rangle$
$\delta(q_0, c, \lambda) = \{[q_1, \lambda]\}$	$\langle q_0, \lambda, q_0 \rangle \rightarrow c \langle q_1, \lambda, q_0 \rangle$ $\langle q_0, \lambda, q_1 \rangle \rightarrow c \langle q_1, \lambda, q_1 \rangle$
$\delta(q_0, c, A) = \{[q_1, A]\}$	$\langle q_0, A, q_0 \rangle \rightarrow c \langle q_1, A, q_0 \rangle$ $\langle q_0, A, q_1 \rangle \rightarrow c \langle q_1, A, q_1 \rangle$
$\delta(q_1, b, A) = \{[q_1, \lambda]\}$	$\langle q_1, A, q_0 \rangle \rightarrow b \langle q_1, \lambda, q_0 \rangle$ $\langle q_1, A, q_1 \rangle \rightarrow b \langle q_1, \lambda, q_1 \rangle$ $\langle q_0, \lambda, q_0 \rangle \rightarrow \lambda$ $\langle q_1, \lambda, q_1 \rangle \rightarrow \lambda$

□

PDA 中的计算和相关联的文法 G 中的推导之间的关系, 采用例 7.3.1 中的 PDA 和文法展示出来。推导以应用规则 S 开始; 后面的步骤根据 M' 中处理相应的输入符号来进行。最左边的变量的第一个部分包含计算中的状态。最右边的变量的第三个部分包含接收状态, 而计算将终止在该状态。连接变量的第二个部分可以获得栈。

$$\begin{array}{ll}
[q_0, aacbb, \lambda] & S \Rightarrow \langle q_0, \lambda, q_1 \rangle \\
\vdash [q_0, acbb, A] & \Rightarrow a \langle q_0, A, q_1 \rangle \\
\vdash [q_0, cbb, AA] & \Rightarrow aa \langle q_0, A, q_1 \rangle \langle q_1, A, q_1 \rangle \\
\vdash [q_1, bb, AA] & \Rightarrow aac \langle q_1, A, q_1 \rangle \langle q_1, A, q_1 \rangle \\
\vdash [q_1, b, A] & \Rightarrow aacb \langle q_1, \lambda, q_1 \rangle \langle q_1, A, q_1 \rangle \\
& \Rightarrow aacb \langle q_1, A, q_1 \rangle \\
\vdash [q_1, \lambda, \lambda] & \Rightarrow aacbb \langle q_1, \lambda, q_1 \rangle \\
& \Rightarrow aacbb
\end{array}$$

变量 $\langle q_0, \lambda, q_1 \rangle$ 可以通过应用 S 规则获得。这表明从状态 q_0 到状态 q_1 的不改变栈的计算就是所需要的计算。后面规则的应用给出了需要一个从状态 q_0 到状态 q_1 的从栈中删除 A 的计算的信号。第4条规则的应用显示了当 δ 中含有不从栈中删除符号的转换的时候, 需要增加转换到 M 中的必要性。规则 $\langle q_0, A, q_1 \rangle \rightarrow c \langle q_1, A, q_1 \rangle$ 表示一个处理符号 c , 但并不将 A 从栈顶删除的计算。

236

现在, 我们已经为证明 PDA 接收的语言是上下文无关的做好了准备。将结果和定理 7.3.1 结合起来就可以得到, 上下文无关规则产生的串和下推自动机接收的串是等价的。

定理 7.3.2 设 M 是一个 PDA。那么存在一个上下文无关的文法 G , 且 $L(G) = L(M)$ 。

前面叙述的从扩展 PDA M' 构造出来的文法 G 和 M 是等价的。必须说明存在一个推导 $S \Rightarrow^* w$, 当且仅当对某些 $q_j \in F$, 有 $[q_0, w, \lambda] \vdash [q_j, \lambda, \lambda]$ 。下面的引理 7.3.3 和引理 7.3.4 将展示 G 中的推导和 M' 中的计算之间的对应关系。

引理 7.3.3 如果 $\langle q_i, A, q_j \rangle \Rightarrow^* w$, 其中 $w \in \Sigma^*$, $A \in \Gamma \cup \{\lambda\}$, 那么 $[q_i, w, A] \vdash [q_j, \lambda, \lambda]$ 。

证明: 通过对终结符号串的推导长度进行归纳来证明本引理。其中的终结符号串是由具有 $\langle q_i, A, q_j \rangle$ 形式的变量推导出来的。归纳的基础是串的推导只包含一个简单规则的应用。空串是能够通过应用一个规则推导出来的唯一的串。推导为 $\langle q_i, \lambda, q_i \rangle \Rightarrow \lambda$, 它应用了第4类规则。在状态 q_i 的空计算产生 $[q_i, \lambda, \lambda] \vdash [q_i, \lambda, \lambda]$, 这就是所需要的。

假设只要 $\langle q_i, A, q_i \rangle \Rightarrow^* v$, 就存在计算 $[q_i, v, A] \vdash [q_j, \lambda, \lambda]$ 。设 w 是终结符号串, 它可以从 $\langle q_i, A, q_j \rangle$ 通过 $n+1$ 步推导而获得。推导的第1步由规则2和规则3的应用组成。以第2类规则开始的推导可以写成

$$\begin{aligned}
\langle q_i, A, q_j \rangle &\Rightarrow u \langle q_k, B, q_j \rangle \\
&\stackrel{n}{\Rightarrow} uv = w
\end{aligned}$$

其中 $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_j \rangle$ 是 G 中的一个规则。根据归纳假设, 存在与推导 $\langle q_k, B, q_j \rangle \stackrel{n}{\Rightarrow} v$ 相对应的计算 $[q_k, v, B] \vdash [q_j, \lambda, \lambda]$ 。

G 中的规则 $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_j \rangle$ 由转换 $[q_k, B] \in \delta(q_i, u, A)$ 产生。将这个转换和计算结合起来, 根据归纳假设可得

$$\begin{aligned}
[q_i, uv, A] &\vdash [q_k, v, B] \\
&\vdash [q_j, \lambda, \lambda]
\end{aligned}$$

237

如果推导的第一步是第3类规则, 则推导可以写成

$$\begin{aligned}
\langle q_i, A, q_j \rangle &\Rightarrow u \langle q_k, B, q_m \rangle \langle q_m, A, q_j \rangle \\
&\stackrel{n}{\Rightarrow} w.
\end{aligned}$$

相应的计算可以从转换 $[q_k, BA] \in \delta(q_i, u, A)$ 中并使用两次归纳假设构造出来。

引理 7.3.4 如果 $[q_i, w, A] \vdash [q_j, \lambda, \lambda]$, 其中 $A \in \Gamma \cup \{\lambda\}$, 那么存在推导 $\langle q_i, A, q_j \rangle \Rightarrow^* w$ 。

证明: 从格局 $[q_i, \lambda, \lambda]$ 得到的空计算是 M 不使用任何转换所能得到的唯一计算。相应的推导包含规则 $\langle q_i, \lambda, q_i \rangle \rightarrow \lambda$ 的简单的应用。

假设每一个计算 $[q_i, v, A] \vdash [q_j, \lambda, \lambda]$ 在 G 中都有一个相对应的推导 $\langle q_i, A, q_j \rangle \Rightarrow v$ 。考虑长度为 $n+1$ 的计算。以非扩展转换开始的指定形式的计算可以写成如下:

$$\begin{aligned} & [q_i, w, A] \\ & \vdash [q_k, v, B] \\ & \vdash [q_j, \lambda, \lambda] \end{aligned}$$

其中 $w = uv$ 并且 $[q_k, B] \in \delta(q_i, u, A)$ 。根据归纳假设, 存在推导 $\langle q_k, B, q_j \rangle \Rightarrow v$ 。第一个转换产生 G 中的规则 $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_j \rangle$ 。所以从 $\langle q_i, A, q_j \rangle$ 到 w 的一个推导可以通过

$$\begin{aligned} & \langle q_i, A, q_j \rangle \Rightarrow u \langle q_k, B, q_j \rangle \\ & \Rightarrow uv \end{aligned}$$

来实现。 M' 中的以扩展转换 $[q_i, BA] \in \delta(q_i, u, A)$ 开始的计算具有下面的形式

$$\begin{aligned} & [q_i, w, A] \\ & \vdash [q_k, v, BA] \\ & \vdash [q_m, y, A] \\ & \vdash [q_j, \lambda, \lambda] \end{aligned}$$

其中 $w = uv$ 并且 $v = xy$ 。规则 $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_m \rangle \langle q_m, A, q_j \rangle$ 由计算中的第一个转换产生。根据归纳假设, G 中包含推导

$$\begin{aligned} & \langle q_k, B, q_m \rangle \Rightarrow x \\ & \langle q_m, A, q_j \rangle \Rightarrow y \end{aligned}$$

将这些推导和前面的规则结合起来就能够从 $\langle q_i, A, q_j \rangle$ 中获得 w 的推导。 ■

定理 7.3.2 的证明: 设 w 是 $L(G)$ 中的任意的串, 它具有推导 $S \Rightarrow \langle q_0, \lambda, q_j \rangle \Rightarrow w$ 。根据引理 7.3.3, 存在计算 $[q_0, w, \lambda] \vdash [q_j, \lambda, \lambda]$, 这表明 M' 接收串 w 。

相反, 如果 $w \in L(M) = L(M')$, 那么存在计算 $[q_0, w, \lambda] \vdash [q_j, \lambda, \lambda]$, 该计算接收 w 。引理 7.3.4 证明了 G 中存在相应的推导 $\langle q_0, \lambda, q_j \rangle \Rightarrow w$ 。因为 q_j 是接收状态, 所以 G 中包含规则 $S \rightarrow \langle q_0, \lambda, q_j \rangle$ 。在文法 G 中, 用此规则开始前面的推导将产生 w 。 ■

7.4 上下文无关语言的泵引理

定理 6.6.3 是正则表达式的泵引理。它说明了正则语言中的一个足够长的串一定具有一个子串, 该子串重复出现任意多次所得到的结果串仍然在该语言中。在本节中, 我们将给出上下文无关语言的泵引理。然而对于上下文无关语言, 泵抽取两个类似的子串。能够产生具有乔姆斯基范式文法的上下文无关语言的这种能力, 提供了证明泵引理所需要的结果。

证明泵引理过程中有两个里程碑。利用乔姆斯基范式文法的规则可以构造推导树。使用推导树的性质, 可以获得数字 k , 它满足:

1. 任何长度大于或等于 k 的串的推导一定具有一个递归的子推导 $A \Rightarrow vAx$, 其中 $v, x \in \Sigma^+$, 并且
2. 串 v 和 x 能够同时从 z 中泵出来, 且结果串仍然在该语言中。

二叉树的叶子数目和深度之间的关系用来实现第一个里程碑, 递归子推导的重复确定了后者。乔姆斯基范式文法的推导树的深度和串长度之间关系在引理 7.4.1 中给出, 并且在推论 7.4.2 中进行了重述。

引理 7.4.1 设 G 是具有乔姆斯基范式的上下文无关的文法, $A \Rightarrow w$ 是具有推导树 T 的推导, 其中 $w \in \Sigma^+$ 。如果 T 的深度是 n , 那么 $\text{length}(w) \leq 2^{n-1}$ 。

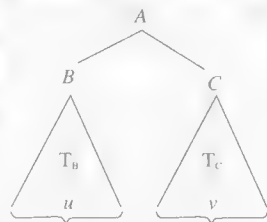
证明: 通过对产生终结字符串的推导树的深度进行归纳来证明本引理。因为 G 是乔姆斯基范式, 推导树的深度为 1 表示产生终结符号串必须是如右所示的两种形式中的一种。在任何一种情况中, 推导串的长度都小于或者等于 $2^0 = 1$, 满足条件。

假设该性质对所有深度小于或者等于 n 的推导树都成立。设 $A \Rightarrow w$ 是推导树 T 的一个长

$$\begin{array}{c} S \\ | \\ \lambda \end{array} \quad \begin{array}{c} A \\ | \\ a \end{array}$$

度为 $n+1$ 的推导。因为文法是乔姆斯基范式, 所以推导可以写成 $A \Rightarrow BC \Rightarrow uv$, 其中 $B \Rightarrow u$, $C \Rightarrow v$ 以及 $w = uv$ 。 $A \Rightarrow w$ 的推导树可以由 T_B 和 T_C 构造, $B \Rightarrow u$ 和 $C \Rightarrow v$ 的推导树如右图所示。树 T_B 和 T_C 的深度都小于或等于 n 。根据归纳假设, $\text{length}(u) \leq 2^{n-1}$ 且 $\text{length}(v) \leq 2^{n-1}$ 。因此 $\text{length}(w) = \text{length}(uv) \leq 2^n$ 。 ■

推论 7.4.2 设 $G = (V, \Sigma, P, S)$ 是一个满足乔姆斯基范式的上下文无关文法, 并且 $S \Rightarrow w$ 是 $w \in L(G)$ 的一个推导。如果 $\text{length}(w) \geq 2^n$, 那么推导树的深入至少是 $n+1$ 。



定理 7.4.3 (上下文无关语言的泵引理) 设 L 是上下文无关的语言。存在一个依赖于 L 的数 k , 使得任何串 $z \in L$, $\text{length}(z) > k$, 都能够写成 $z = uvwxy$, 其中

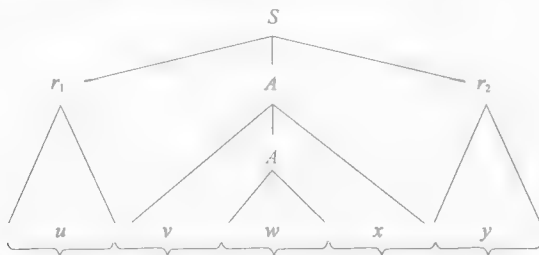
- i) $\text{length}(vwx) \leq k$
- ii) $\text{length}(v) + \text{length}(x) > 0$
- iii) 对任意的 $i \geq 0$, 有 $uv^iwx^iy \in L$ 。

证明: 设 $G = (V, \Sigma, P, S)$ 是产生语言 L 的乔姆斯基范式文法, 并设 $k = 2^n, n = \text{card}(V)$ 。下面说明所有长度大于等于 k 的串都能够分解以满足泵引理的条件。设 $z \in L(G)$ 是这样的一个串, $S \Rightarrow z$ 是 G 中的一个推导。由推论 7.4.2, 在 $S \Rightarrow z$ 的推导树中, 存在长度至少是 $n+1 = \text{card}(V) + 1$ 的路径。

设 p 是推导树中从根 S 到叶节点的最大长度的路径。那么 p 至少包含 $n+2$ 个节点, 除叶节点外, 所有的节点都标记有一个变量, 叶节点标记为一个终结符号。鸽巢原理保证了至少有一个变量 A 在这条具有 $n+2$ 个节点的路径中出现两次。虽然 A 可能在路径中出现超过两次, 但是我们只关心在 p 中最后的出现和紧挨着最后的出现。

[240]

将推导树中路径的性质转移到子推导中, z 的推导可以描述如下:



其中 $z = uvwxy$ 。推导 $S \Rightarrow r_1 A r_2$ 产生挨着最后一次出现的变量 A 。在应用推导 $A \Rightarrow w$ 之前, 子推导 $A \Rightarrow vAx$ 可以被去掉或者重复任意次数。推导结果产生串 $uv^iwx^iy \in L(G) = L$ 。

现在将说明, 这种分解能够满足泵引理的条件 (i) 和条件 (ii)。子推导 $A \Rightarrow vAx$ 必须从形式为 $A \Rightarrow BC$ 的规则开始。变量 A 的第二次出现可以从 B 或者 C 推导出来。如果它从 B 推导出来, 那么推导可以写成:

$$\begin{aligned} A &\Rightarrow BC \\ &\stackrel{*}{\Rightarrow} vAsC \\ &\stackrel{*}{\Rightarrow} vAst \\ &= vAx \end{aligned}$$

串 t 是非空的, 因为它是由乔姆斯基范式文法中的一个变量推导而来的, 该变量不是文法的开始符号之后, x 也是非空的。如果 A 的第二次出现来自变量 C , 那么一个类似的讨论可以表明 v 也一定不是空串。

在路径 p 中, A 的最后两次出现之间的子路径的长度最多是 $n+2$ 。由推导 $A \Rightarrow vwx$ 产生的推导树的深度最多是 $n+1$ 。由引理 7.4.1, 串 vwx 可以由长度小于等于 $k = 2^n$ 的推导获得。 ■

和正则语言的泵引理一样, 泵引理为表明一个语言不是上下文无关的提供了一个工具。根据泵引

[241] 理, 上下文无关文法中的每一个足够长的串都可以泵出子串。所以我们通过找到一个不能分解成满足定理 7.4.3 描述的 $uvwx$ 形式的串, 来说明一个语言不是上下文无关的。

例 7.4.1 语言 $L = \{a^i b^j c^k \mid i \geq 0\}$ 不是上下文无关的。假设 L 是上下文无关的。根据定理 7.4.1, 串 $z = a^k b^k c^k$, 其中 k 是满足泵引理的数, 能够分解成满足重复性质的子串 $uvwx$ 。考虑子串 v 和 x 的可能性。如果其中一个包含多于两种的终结符号, 那么 $uv^i wx^i y$ 包含一个在 a 之前的 b 或者在 b 之前的 c 。任何一种情况, 结果串都不在 L 中。

根据前面的观察, v 和 x 必须是 a^* 、 b^* 和 c^* 中的一种子串。因为 v 和 x 中最多只有一个空, 所以 $uv^i wx^i y$ 将增加一种, 最多两种, 但不可能是三种终结符号的长度。这表明 $uv^i wx^i y$ 不在 L 中。所以, 不存在 $a^k b^k c^k$ 的一个分解, 使得其满足泵引理; 因此, L 不是上下文无关的。□

例 7.4.2 语言 $L = \{a^i b^j a^k \mid i, j \geq 0\}$ 不是上下文无关的。设 k 是泵引理中指定的数且 $z = a^k b^k a^k$ 。假设存在 z 的一个分解 $uvwx$, 该分解满足泵引理。条件 (ii) 要求 wx 的长度最多是 k 。这表明 wx 是一个只包含一种终结符的串或者两个这样的串的连接。即

i) $wx \in a^*$ 或 $wx \in b^*$, 或者

ii) $wx \in a^* b^*$ 或 $wx \in b^* a^*$

根据类似例 7.4.1 中的讨论, 子串 v 和 x 必须只包含一种终结符。泵作用 v 和 x , 将只在 z 的一个子串中增加 a 或者 b 的数目。因此 z 不存在满足泵引理条件的分解, 于是我们得出结论, L 不是上下文无关的。□

例 7.4.3 语言 $L = \{w \in a^* \mid \text{length}(w) \text{ 是素数}\}$ 不是上下文无关的。假设 L 是上下文无关的, n 是大于 k 的素数, k 是定理 7.4.3 中的常数。串 a^n 必须具有一个满足泵引理条件的分解 $uvwx$ 。设 $m = \text{length}(u) + \text{length}(w) + \text{length}(y)$, 任意串 $uv^i wx^i y$ 的长度为 $m + i(n - m)$ 。

[242] 特别地, $\text{length}(wv^{n+1}wx^{n+1}y) = m + (n+1)(n-m) = n(n-m+1)$ 。在前面的积中的两个项, 都是大于 1 的自然数。因此, $wv^{n+1}wx^{n+1}y$ 的长度不是素数, 该串不在 L 中。所以, L 不是上下文无关的。□

7.5 上下文无关语言的封闭性

我们利用上下文无关文法的适应性来确定上下文无关语言集的封闭结果。保持上下文无关语言的操作提供了另一个证明语言是上下文无关的工具。这些操作, 与前面的泵引理结合, 也可以用来说明某些语言不是上下文无关的。

定理 7.5.1 上下文无关语言族在并、连接和克林星 (Kleene star) 操作下是封闭的。

证明: 设 L_1 和 L_2 是分别由 $G_1 = (V_1, \Sigma_1, P_1, S_1)$ 和 $G_2 = (V_2, \Sigma_2, P_2, S_2)$ 产生的上下文无关文法。假设变量的集合 V_1 和 V_2 是不相交的。由于我们可以改变变量的名字, 因此该假设没有对文法增加任何的约束。

从 G_1 和 G_2 构造一个上下文无关文法来证明要求的封闭性。

并: 定义 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$ 。串 w 在 $L(G)$ 中, 当且仅当存在推导 $S \Rightarrow_{G_1}^i w, i = 1$ 或 2 。所以 w 在 L_1 或者 L_2 中。另一方面, 任何推导 $S_i \Rightarrow_{G_1}^i w$ 在 G 中都能够从 $S \rightarrow S_i$ 开始产生串 w 。

连接: 定义 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$ 。开始符号在 G_1 和 G_2 中都初始化了接收 w 的推导。在 G 中的一个终结符号串的最左推导具有 $S \Rightarrow S_1 S_2 \Rightarrow u S_2 \Rightarrow uv$ 的形式, 其中 $u \in L_1, v \in L_2$ 。 u 的推导利用 P_1 中的规则, v 的推导利用 P_2 中的规则。所以 $L(G) \subseteq L_1 L_2$ 。相反, 通过观察可以确定, 每一个在 $L_1 L_2$ 中的串 w 都能够写成 uv , 其中 $u \in L_1, v \in L_2$ 。推导 $S_1 \Rightarrow_{G_1}^i u$ 和 $S_2 \Rightarrow_{G_2}^j v$, 与 G 中的 S 规则, 在 G 中可以产生 w 。

克林星: 定义 $G = (V_1, \Sigma_1, P_1 \cup \{S \rightarrow S_1 S \mid \lambda\}, S)$ 。 G 中的 S 规则产生任意数目的 S_1 的副本。每一个这种副本依次开始 L_1 中串的推导。连接任意多个 L_1 中的串将产生 L_1^* 。■

定理 7.5.1 展示了上下文无关语言集的正的封闭的结果。可以给一个简单的例子说明上下文无关

语言在交下是不封闭的。最后,我们结合并的封闭性和交来得到一个补也是不封闭的结果。

定理 7.5.2 上下文无关语言在交和补下是不封闭的。

证明: 交: 设 $L_1 = \{a^i b^j c^j \mid i, j \geq 0\}$ 和 $L_2 = \{a^i b^j c^i \mid i, j \geq 0\}$. L_1 和 L_2 是分别由 G_1 和 G_2 产生的上下文无关语言。 [243]

$$\begin{array}{ll} G_1: S \rightarrow BC & G_2: S \rightarrow AB \\ B \rightarrow aBb \mid \lambda & A \rightarrow aA \mid \lambda \\ C \rightarrow cC \mid \lambda & B \rightarrow bBc \mid \lambda \end{array}$$

L_1 和 L_2 的交是集合 $\{a^i b^j c^j \mid i \geq 0\}$, 由例 7.4.1 知该集合不是上下文无关的。

补: 设 L_1 和 L_2 是任意的两个上下文无关的语法。如果上下文无关语言在补下是封闭的, 那么由定理 7.5.1, 语言

$$L = \overline{L_1} \cup \overline{L_2}$$

也是上下文无关的。根据德摩根定律, $L = L_1 \cap L_2$ 。这表明上下文无关语言在交下是封闭的, 这与第 1 部分的结果相矛盾。 ■

第 6 章的练习 9 显示了正则语言和上下文无关语言的交不是正则的。语言和下推自动机的对应关系可以用来确定正则和上下文无关语言的交的封闭性。

设 R 是被 DFA N 接收的正则语言, L 是被 PDA M 接收的上下文无关语言。我们将说明 $R \cap L$ 是上下文无关的, 方法是用一个模拟 N 和 M 的操作的 PDA 来构造该语言。复合自动机的状态是由 M 和 N 的状态组成的有序对。

定理 7.5.3 设 R 是正则语言, L 是上下文无关语言。语言 $R \cap L$ 是上下文无关的。

证明: 设 $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$ 是接收 R 的 DFA, 而 $M = (Q_M, \Sigma_M, \Gamma, \delta_M, q_0, F_M)$ 是接收语言 L 的 PDA。自动机 N 和 M 相结合构成 PDA

$$M' = (Q_M \times Q_N, \Sigma_M \cup \Sigma_N, \Gamma, \delta, [p_0, q_0], F_M \times F_N)$$

它接收 $R \cap L$ 。 M' 的转换函数定义为“同时运行自动机 M 和 N ”。有序对的第一个部分是 M 进入的状态序列, 第二部分是 N 进入的状态序列。转换函数 M' 定义为

$$\text{i) } \delta([p, q], a, A) = \{[[p', q'], B] \mid [p', B] \in \delta_M(p, a, A), \text{ 并且 } \delta_N(q, a) = q'\}$$

$$\text{ii) } \delta([p, q], \lambda, A) = \{[[p', q'], B] \mid [p', B] \in \delta_M(p, \lambda, A)\}$$

DFA 的每一个转换处理一个输入符号, 然而 PDA 的有些转换并不处理输入。条件 (ii) 引入的转换模拟了 PDA 不处理输入符号的转换的动作。

串 w 被 M' 接收, 如果存在计算:

$$[[p_0, q_0], w, \lambda] \vdash^* [[p_i, q_j], \lambda, \lambda]$$

其中, p_i 和 q_j 分别是 M 和 N 的终止状态。 [244]

可得出 $L(N) \cap L(M) \subseteq L(M')$, 因为存在下面的计算

$$[[p_0, q_0], w, \lambda] \vdash^* [[p_i, q_j], u, \alpha]$$

只要

$$[p_0, w, \lambda] \vdash^* [p_i, u, \alpha] \text{ 并且 } [q_0, w] \vdash^* [q_j, u]$$

是 M 和 N 中的计算。证明可以通过对 PDA M 上的转换数进行归纳。

递归的基础是 M 中的空计算。计算终止时, $p_i = p_0$, $u = w$ 并且 M 含有一个空栈。 N 中以原串终止的唯一的计算是空计算; 所以 $q_j = q_0$ 。在复合机器中, 相对应的计算是 M' 中的空计算。

假设对于 M' 中所有长度为 n 的计算结果都成立。设

$$[p_0, w, \lambda] \vdash^* [p_i, u, \alpha] \text{ 并且 } [q_0, w] \vdash^* [q_j, u]$$

分别是 PDA 和 DFA 中的计算。 M 中的计算可以写为:

$$\begin{array}{l} [p_0, w, \lambda] \\ \vdash^* [p_k, v, \beta] \\ \vdash^* [p_i, u, \alpha] \end{array}$$

其中, $v = u$ 或者 $v = au$ 。为了展示存在计算 $[[p_0, q_0], w, \lambda] \vdash^* [[p_i, q_j], u, \alpha]$, 下面分别讨论 v 的每一种可能情况。

情况 1: $v = u$ 。在这种情况下, M 中的计算的最终转换并不处理输入符号。 M 中的计算是通过形式为 $[p_i, B] \in \delta_M(p_k, \lambda, A)$ 的转换完成的。这个转换在 M' 中产生 $[[p_i, q_i], B] \in \delta([p_k, q_j], \lambda, A)$ 。计算:

$$\begin{aligned} [[p_0, q_0], w, \lambda] &\vdash^* [[p_k, q_j], v, \beta] \\ &\vdash^* [[p_i, q_j], v, \alpha] \end{aligned}$$

可以从归纳假设和 M' 中前面的转换获得。

情况 2: $v = au$ 。 N 中将 w 规约到 u 的计算可以写为:

$$\begin{aligned} [q_0, w] \\ &\vdash^* [q_m, v] \\ &\vdash^* [q_j, u] \end{aligned}$$

[245] 其中, 最后一步应用转换 $\delta_N(q_m, a) = q_j$ 。组合 DFA 和 PDA 对输入符号 a 的转换可以得到 M' 中的转换 $[[p_i, q_j], B] \in \delta([p_k, q_m], a, A)$ 。将这个转换应用到计算的结果, 根据归纳假设可以得到

$$\begin{aligned} [[p_0, q_0], w, \lambda] &\vdash^* [[p_k, q_m], v, \beta] \\ &\vdash^* [[p_i, q_j], u, \alpha] \end{aligned}$$

另一方面, $L(M') \subseteq L(N) \cap L(M)$ 可以通过对 M' 中计算的长度进行归纳而得证。证明将留做练习。■

定理 7.5.2 使用了德摩根定律说明了上下文无关语言在补下是不封闭的。下面的例子给出一个清楚展示这个性质的文法。

例 7.5.1 语言 $L = \{ww \mid w \in \{a, b\}^*\}$ 不是上下文无关的, 但 \bar{L} 是上下文无关的。首先, 我们将通过反正法来说明 L 不是上下文无关的。假设 L 是上下文无关的。那么, 由定理 7.5.3

$$L \cap a^* b^* a^* b^* = \{a^i b^j a^i b^j \mid i, j \geq 0\}$$

是上下文无关的。然而, 在例 7.4.2 中已经证明了该语言不是上下文无关的, 与我们的假设矛盾。

为了说明 \bar{L} 是上下文无关的, 需要构造两个上下文无关文法 G_1 和 G_2 , 使得 $L(G_1) \cup L(G_2) = \bar{L}$ 。

$$\begin{aligned} G_1: S &\rightarrow aA \mid bA \mid a \mid b & G_2: S &\rightarrow AB \mid BA \\ A &\rightarrow aS \mid bS & A &\rightarrow ZAZ \mid a \\ & & B &\rightarrow ZBZ \mid b \\ & & Z &\rightarrow a \mid b \end{aligned}$$

文法 G_1 产生 $\{a, b\}$ 上的长度为奇数的串, 所有这些串都在 \bar{L} 中。 G_2 产生 \bar{L} 中的长度为偶数的串。这样的串可以写成 $u_1 x v_1 u_2 y v_2$, 其中 $x, y \in \Sigma$ 并且 $x \neq y$, u_1, v_1, u_2 以及 $v_2 \in \Sigma^*$, $\text{length}(u_1) = \text{length}(u_2)$ 且 $\text{length}(v_1) = \text{length}(v_2)$ 。即, x 和 y 是出现在子串中相同位置的不同的符号, 这个子串分别组成了 $u_1 x v_1 u_2 y v_2$ 的前半部分和后半部分。因为 u 和 v 是任意的串, 所以这个性质可以写成 $u_1 x p q y v_2$, 其中 $\text{length}(p) = \text{length}(u_1)$, $\text{length}(q) = \text{length}(v_2)$ 。 G_2 中的递归变量产生的恰好是这些串的集合。□

7.6 练习

1. 设 M 是 PDA, 其定义如下

$$\begin{aligned} Q &= \{q_0, q_1, q_2\} & \delta(q_0, a, \lambda) &= \{[q_0, A]\} \\ \Sigma &= \{a, b\} & \delta(q_0, \lambda, \lambda) &= \{[q_1, \lambda]\} \\ \Gamma &= \{A\} & \delta(q_0, b, A) &= \{[q_2, \lambda]\} \\ F &= \{q_1, q_2\} & \delta(q_1, \lambda, A) &= \{[q_1, \lambda]\} \\ & & \delta(q_2, b, A) &= \{[q_2, \lambda]\} \\ & & \delta(q_2, \lambda, A) &= \{[q_2, \lambda]\} \end{aligned}$$

- a) 描述 M 接收的语言。
 - b) 给出 M 的状态图。
 - c) 跟踪 M 对串 aab 、 abb 和 aba 的所有计算。
 - d) 说明 $aabb$, $aabb \in L(M)$ 。
2. 设 M 是例 7.1.3 中的 PDA
- a) 给出 M 的转换表。
 - b) 跟踪 M 对 ab 、 abb 和 $abbb$ 的所有计算。
 - c) 说明 $aaaa$, $baab \in L(M)$ 。
 - d) 说明 aaa , $ab \notin L(M)$ 。
3. 构造接收下面的每一个语言的自动机
- a) $\{a^i b^j \mid 0 \leq i \leq j\}$
 - b) $\{a^i c^j b^k \mid i, j \geq 0\}$
 - c) $\{a^i b^j c^k \mid i + k = j\}$
 - d) $\{w \mid w \in \{a, b\}^*, w \text{ 中 } a \text{ 的数目是 } b \text{ 的两倍}\}$
 - e) $\{a^i b^j \mid i \geq 0\} \cup a^* \cup b^*$
 - f) $\{a^i b^j c^k \mid i = j \text{ 或 } j = k\}$
 - g) $\{a^i b^j \mid i \neq j\}$
 - h) $\{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$
 - i) $\{a^{i+j} b^i c^j \mid i, j > 0\}$
 - j) $\{a, b\}$ 上的回文串的集合。

4. 构造一个接收下面语言的只含有两个栈元素的 PDA

$$\{wdw^R \mid w \in \{a, b, c\}^*\}$$

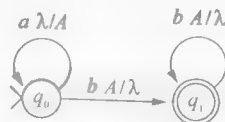
247

5. PDA 空栈接收语言 $\{a^i b^{j+i} \mid 0 \leq j \leq i\}$, 给出其状态图, 并解释栈符号在 M 的计算中的作用。跟踪 M 对输入 $aabb$ 和 $aaaabb$ 的计算。

6. 自动机 M

终结状态空栈接收语言 $\{a^i b^j \mid i > 0\}$ 。

- a) 给出空栈接收 L 的 PDA 的状态图。
- b) 给出终结状态接收 L 的 PDA 的状态图。



7. 设 L 是语言 $\{w \in \{a, b\}^* \mid w \text{ 有这样一个前缀, 该前缀之中 } b \text{ 的个数多于 } a \text{ 的个数}\}$ 。例如 baa , $abba$, $abbaaa \in L$, 但是 aab , $aabbab \notin L$
- a) 构造一个终结状态接收 L 的 PDA。
 - b) 构造一个空栈接收 L 的 PDA。
8. 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是被终结状态和空栈接收 L 的 PDA。证明存在一个只被终结状态接收 L 的 PDA。
9. 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是被终结状态和空栈接收 L 的 PDA。证明存在一个只被空栈接收 L 的 PDA。
10. 设 $L = \{a^{2i} b^i \mid i \geq 0\}$
- a) 构造一个 PDA M_1 , 使得 $L(M_1) = L$ 。
 - b) 构造一个原子 PDA M_2 , 使得 $L(M_2) = L$ 。
 - c) 构造一个扩展 PDA M_3 , 使得 $L(M_3) = L$, 并且它比 M_1 具有较少的状态。
 - d) 跟踪 (a)、(b) 和 (c) 中自动机对输入 aab 的计算。
11. 设 $L = \{a^{2i} b^{3i} \mid i \geq 0\}$
- a) 构造一个 PDA M_1 , 使得 $L(M_1) = L$ 。
 - b) 构造一个原子 PDA M_2 , 使得 $L(M_2) = L$ 。
 - c) 构造一个扩展 PDA M_3 , 使得 $L(M_3) = L$, 并且它比 M_1 具有较少的状态。
 - d) 跟踪 (a)、(b) 和 (c) 中自动机对输入 $aabbb$ 的计算。

12. 利用定理 7.3.1 中的方法构造一个接收下面格立巴赫范式文法的语言的 PDA。

$$S \rightarrow aABA \mid aBB$$

$$A \rightarrow bA \mid b$$

$$B \rightarrow cB \mid c$$

248

13. 设 G 是格立巴赫范式文法, M 是从 G 中构造的 PDA。证明如果 M 中具有转换 $[q_0, u, \lambda] \vdash [q_1, \lambda, w]$, 那么 G 中存在推导 $S \xRightarrow{*} uw$ 。这完成了定理 7.3.1 的证明。

14. 设 M 是 PDA

$$Q = \{q_0, q_1, q_2\} \quad \delta(q_0, a, \lambda) = \{[q_0, A]\}$$

$$\Sigma = \{a, b\} \quad \delta(q_0, b, A) = \{[q_1, \lambda]\}$$

$$\Gamma = \{A\} \quad \delta(q_1, b, \lambda) = \{[q_2, \lambda]\}$$

$$F = \{q_2\} \quad \delta(q_2, b, A) = \{[q_1, \lambda]\}$$

- 给出 M 的状态图。
 - 给出 $L(M)$ 的集合的定义。
 - 使用定理 7.3.2, 构造一个产生 $L(M)$ 的上下文无关文法 G 。
 - 跟踪 M 中对 $aabbbb$ 的计算。
 - 给出 G 中 $aabbbb$ 的推导。
15. 设 M 是例 7.1.1 中的 PDA
- 跟踪 M 中接收 $bcbcb$ 的计算。
 - 利用定理 7.3.2 的计算, 构造一个接收 $L(M)$ 的文法 G 。
 - 给出 G 中 $bcbcb$ 的推导。
- * 16. 定理 7.3.2 给出了构造产生扩展 PDA 接收的的语言的文法的方法。PDA 的转换最多将两个变量压入栈中。推广从任意的扩展 PDA 构造文法的过程。证明得到的文法能够产生 PDA 接收的语言。
17. 使用泵引理证明下面的各个语言不是上下文无关的。
- $\{a^k \mid k \text{ 是完全平方数}\}$
 - $\{a^i b^j c^i d^j \mid i, j \geq 0\}$
 - $\{a^i b^{2i} a^i \mid i \geq 0\}$
 - $\{a^i b^j c^k \mid 0 < i < j < k < zi\}$
 - $\{ww^R w \mid w \in \{a, b\}^*\}$
 - 无穷串的有限长度的前缀构成的集合

$$abaabaaabaaaab \cdots ba^n ba^{n+1} b \dots$$

- 证明语言 $L_1 = \{a^i b^{2i} c^j \mid i, j \geq 0\}$ 是上下文无关的。
 - 证明语言 $L_2 = \{a^i b^j c^{2i} \mid i, j \geq 0\}$ 是上下文无关的。
 - 证明 $L_1 \cap L_2$ 不是上下文无关的。
19. a) 证明语言 $L_1 = \{a^i b^j c^i d^j \mid i, j \geq 0\}$ 是上下文无关的。
 b) 证明语言 $L_2 = \{a^i b^j c^i d^k \mid i, j, k \geq 0\}$ 是上下文无关的。
 c) 证明 $L_1 \cap L_2$ 不是上下文无关的。
20. 设 L 是 $\{a, b\}$ 上的所有 a 和 b 数目相同的串构成的语言。说明 L 满足泵引理。即, 说明每一个长度大于等于 k 的串 z , 都存在一个满足泵引理的分解。
21. 设 M 是 PDA。证明存在一个决定下面是否正确的计算过程:
- $L(M)$ 是空。
 - $L(M)$ 是有限的。
 - $L(M)$ 是无限的。

249

- * 22. 文法 $G = (V, \Sigma, P, S)$ 称作是线性的 (linear), 如果 G 中每一个规则都具有下面的形式;

$$A \rightarrow u$$

$$A \rightarrow uBv$$

其中 $u, v \in \Sigma^*$, 并且 $A, B \in V$ 。如果产生语言的文法是线性的, 那么该语言称作是线性的。证明下面对线性语言的泵引理。

设 L 是线性语言。那么存在一个常数 k , 对所有的 $z \in L$ 有 $\text{length}(z) \geq k$, z 可以写成 $z = uvwxy$, 并且满足下面条件

- i) $\text{length}(uvxy) \leq k$,
- ii) $\text{length}(vx) > 0$, 并且
- iii) $uv^iwx^iy \in L, i \geq 0$ 。

- 23. a) 构造一个接收 $\{a, b\}$ 上的含奇数个 a 的串的 DFA N 。
- b) 构造一个接收 $\{a^{3i}b^i \mid i \geq 0\}$ 的 PDA M 。
- c) 利用定理 7.5.3 中的方法构造一个接收 $L(N) \cap L(M)$ 的 PDA M' 。
- d) 跟踪 N 、 M 和 M' 中接收 $aaab$ 的计算。

- 24. 设 $G = (V, \Sigma, P, S)$ 是上下文无关的文法。定义一个扩展 PDA M

$$\begin{aligned} Q &= \{q_0\} & \delta(q_0, \lambda, \lambda) &= \{[q_0, S]\} \\ \Sigma &= \Sigma_G & \delta(q_0, \lambda, A) &= \{[q_0, w] \mid A \rightarrow w \in P\} \\ \Gamma &= \Sigma_G \cup V & \delta(q_0, a, a) &= \{[q_0, \lambda] \mid a \in \Sigma\} \\ F &= \{q_0\} \end{aligned}$$

证明 $L(M) = L(G)$ 。

- 25. 完成定理 7.5.3 的证明。

[250]

- 26. 证明上下文无关语言集在逆转运算下是封闭的。

- *27. 设 L 是 Σ 上的上下文无关语言, $a \in \Sigma$ 。定义 $er_a(L)$ 为将 L 中的串内出现的 a 都删除后所获得的串的集合。语言 $er_a(L)$ 是从 L 中删除 a 所得到的语言。例如, 如果 $abab$ 、 $bacb$ 和 $aa \in L$, 那么 bb 、 bcb 和 $\lambda \in er_a(L)$ 。证明 $er_a(L)$ 是上下文无关的。提示: 将产生 L 的文法转化为产生 $er_a(L)$ 的文法。

- *28. 在练习 6.19 中已经引入了串同态的记号。设 L 是 Σ 上的上下文无关文法, $h: \Sigma^* \rightarrow \Sigma^*$ 是一个同态。

- a) 证明 $h(L) = \{h(w) \mid w \in L\}$ 是上下文无关的, 也就是说, 上下文无关语言在同态操作下是封闭的。
- b) 利用(a)的结果说明 $er_a(L)$ 是上下文无关的。
- c) 给出一个例子说明一个非上下文无关语言的同态象可能是上下文无关的语言。

- 29. 设 $h: \Sigma^* \rightarrow \Sigma^*$ 是一个同态, L 是 Σ 上的上下文无关语言。证明 $\{w \mid h(w) \in L\}$ 是上下文无关的。换言之, 上下文无关语言族在逆同态象作用下是封闭的。

- 30. 使用同态象和逆同态象说明下面的语言不是上下文无关的。

- a) $\{a^ib^jc^id^j \mid i, j \geq 0\}$
- b) $\{a^ib^{2i}c^{3i} \mid i \geq 0\}$
- c) $\{(ab)^i(bc)^i(ca)^i \mid i \geq 0\}$

参考文献注释

下推自动机在 Oettinger [1961] 中引入。Fischer [1963] 和 Schutzenberger [1963] 中研究了确定型下推自动机, Knuth [1995] 中研究了接收 $LR(k)$ 文法产生的语言的确定型下推自动机。Chomsky [1962]、Evey [1963] 和 Schutzenberger [1963] 发现了上下文无关语言和下推自动机之间的关系。7.5 节中的上下文无关语言的闭包性来自 Bar-Hillel、Perles 和 Shamir [1961] 以及 Scheinberg [1960]。练习 28 和练习 29 的解答可以在 Ginsburg 和 Rose [1963b] 中找到。

上下文无关语言的泵引理来自 Bar-Hillel、Perles 和 Shamir [1961]。Ogden [1968] 中给出了一个更老版本的泵引理。Parikh [1966] 的理论提供了证明不是上下文无关语言的另一个工具。

[251]
[252]

第三部分

可 计 算 性

我们现在开始了解算法的计算能力及其局限性。术语“有效过程”被用来描述我们直觉上认为是可计算的过程。一个“有效过程”包括了一个指令的有限集合以及相应的规格说明书，此规格说明书基于输入，并以指令的执行顺序来编排。指令的执行不需要执行计算的机器或者人有多么的聪明或者富有创造性，它实际上是十分机械化的。有效过程所产生的计算首先需要执行有限数目的指令，然后终止。上述的属性简要描述如下：有效过程是一个确定性的离散过程，并且对于任何可能的输入均会停机。

英国数学家阿兰·图灵于1936年设计了一系列的抽象机，并将这些抽象机用于执行有效计算。图灵机则在这一系列能力逐渐增强的抽象计算机中（包括有限自动机以及下推自动机），达到了一个顶峰。与有限自动机类似，可应用图灵机的指令是由机器当前的状态以及正在读取的符号来决定的。但是图灵机也许会多次读取其输入字符串，并且一条指令也有可能将信息写到存储器上。图灵机执行多次读及写操作的能力使其计算能力得以增强，从而为现代计算机提供了理论模型。

于1936年由逻辑学家阿兰佐·丘奇提出的丘奇—图灵论题，曾断言道：在任意的算法系统中的任何有效计算都能够用一个图灵机完成。丘奇—图灵论题并不应该被认为是给算法系统下定义——因为算法系统仅是其中一个极端狭隘的视角，实际上存在着许多被设计来执行有效计算的系统。况且，谁又能预测出将来还会发展出什么样的形式主义或技术呢？丘奇—图灵论题也并不是认为其他系统不执行算法的计算。实际上，这个论题断言了在任何类似的系统中执行的计算都能够通过设计一个相应的图灵机来完成。也许能够支持丘奇—图灵论题的最有力证据是在论题提出的70年后，没有任何人发现一个反例。对于该理论的表述以及其对于可计算性的含义将在第11章中讨论。

253

通过文法生成的语言，以及这些语言被机器所识别，两者之间的一致性扩展成了图灵机的语言。如果图灵机代表了字符串识别机器的极致的话，那么其相关的文法被认为是最通用的字符串转化系统也就是合理的了。这实际上就是事实，与图灵机相对应的文法称为无限制文法也正是因为对于其规则的应用形式没有任何的限制。为了建立图灵机识别以及无限制文法生成之间的一致性，我们将会展示能够使用无限制文法的推导来对图灵机的计算进行模拟。

既然接受了丘奇—图灵论题，那么算法问题的解决的范围就能够通过图灵机的计算能力来界定。因此，为了证明一个问题没有解，仅需要证明不存在针对该问题的图灵机解决方案。使用这个方法，我们可以了解到，图灵机的停机问题是不能够被确定的。也就是说，没有任何的算法能够确定，对于任意的图灵机 M 和字符串 w ，图灵机 M 是否会在输入字符串为 w 的计算中停机。我们因此可以缩减问题从而确立不可判定性，这种不可判定性主要涉及图灵机计算结果的额外问题，或者使用文法规则的推导的存在性，甚至上下文无关语言的属性。

254

第8章 图灵机

图灵机，于1936年由阿兰·图灵提出，代表了有限状态计算自动机发展的另外一个阶段。提出图灵机的最初目的，主要是为了研究有效计算，但展示出了与现代计算机普遍相关的很多特点。这毫不奇怪，图灵机提供了设计以及开发“程序存储”计算机的一个模型。通过一系列的基本操作，图灵机能够访问以及修改任何存储位置的数据。和计算机的不同之处在于：图灵机为了完成一个计算，在时间上和空间上没有任何限制。

丘奇—图灵论题提出了这样一个论断，任何有效过程均能被一个适当设计的图灵机实现，这将在第11章详细讨论。下面两章将要介绍的关于图灵机体系结构的变种以及其应用，揭示了图灵机计算上的健壮性以及多功能性。

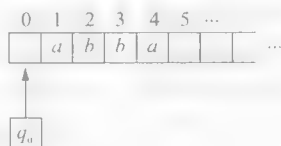
8.1 标准图灵机

图灵机是这样一种有限状态机，在每次状态转换的时候在带上面打印一个符号。带头可以双向运动，从而允许图灵机能够按照要求的次数读取以及操作输入。图灵机的结构与有限状态自动机很相像，只是状态转换函数具有一些额外的特性。

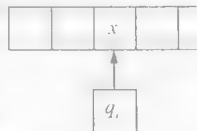
定义 8.1.1 图灵机 (turing machine) 是一个五元组 $M = (Q, \Sigma, \Gamma, \delta, q_0)$, Q 是状态的有限集合, Γ 是一个称为“带字母表”的有限集, Γ 中包含了一个特殊的符号 B 来代表空白, Σ 则是 $\Gamma - \{B\}$ 的一个子集, 称为输入字母表, δ 是从 $Q \times \Gamma$ 到 $Q \times \Gamma \times \{L, R\}$ 的部分函数, 称为转换函数, 并且状态 $q_0 \in Q$ 是一个特殊的状态, 称为初始状态。

图灵机的带有一个左边界, 并且能够向右无限扩展。带上的位置则由自然数来标识, 最左边的位置被标记为 0, 每一个带的位置均包含着带字母表上的一个元素。

任何一个计算的初始都是这样的, 机器的状态是 q_0 , 带头扫描最左边的位置。而输入的则是 Σ^* 中的一个字符串, 从位置 1 开始写在带上。位置 0 以及带剩余的部分都是空白。上图显示了有着 *abba* 输入的图灵机的初始构造。带字母表提供了一些额外的符号, 这些符号可能会在计算中用到。

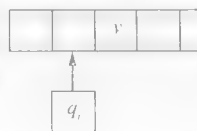


转换往往包含三个动作：更改状态，在带头正在读的方格中写入一个符号，然后移动带头。带头的移动方向由这个转换中的最后一个元素所决定，L 意味着带头往左边移动，而 R 则意味着带头往右移动。图灵机现在的格局以及状态转换 $\delta(q_i, x) = [q_j, y, L]$ 共同产生了新格局。



这个转换将图灵机的状态从 q_i 转换到了 q_j , 用 y 替换了带上原来的符号 x , 并且将带头向左移动了一个方格。图灵机的这种能够使带头向两个方向移动, 以及其处理空白的能力, 使得进行无限连续计算成为可能。

当遇到没有定义其转换的状态或者符号对的时候, 计算随即终止。而某个从位置 0 开始的转换也可能会向左移动, 从而移出带的边界。当上述情况发生的时候, 这次计算会被称为不正常的终结 (terminate abnormally)。当我们说某个计算终止, 我们往往是说其是正常终结的。



在定义 8.1.1 中所提到的图灵机是确定型的, 也就是说, 最多每一个转换与每一个状态以及带符号的组合一一对应。单带确定型图灵机, 若具有上述的初始条件, 则称为标准图灵机 (standard turing machine)。最初的两个例子描述了如何使用图灵机来操作字符串。在使用图灵

机计算开发了一个工具之后,我们将会运用图灵机来接收语言以及计算函数。

例 8.1.1 下面的表格代表了输入字母表为 $\{a, b\}$ 的标准图灵机转换函数:

δ	B	a	b
q_0	q, B, R		
q_1	q_1, B, L	q, b, R	q, a, R
q_2		q, a, L	q_2, b, L

从状态 q_0 开始的转换将带头移动到位置 1 来读取输入。从状态 q_1 开始的转换读取输入字符串,然后交换了符号 a 和符号 b 的位置。从状态 q_2 开始的转换将图灵机返回到了初始位置。

图灵机也能够使用一个状态图来表示。转换函数 $\delta(q_i, x) = \langle q_j, y, d \rangle, d \in \{L, R\}$ 被表示为从 q_i 到 q_j 的一条标记为 x/yd 的弧。右侧的状态图代表了在上面的转换表中所定义的图灵机。□

图灵机的格局包含了状态,带以及带头的位置。在标准图灵机的任意一步计算中,带仅有有限的一部分是非空白符号。一个格局被表示成为 $uq_i vB$, 其中 B 右边的所有位置均是空白,而 uv 则是从带的左边界到 B 使用符号所表示的一个字符串。 uv 中也可能存在空白,但是 uv 中包含了所有的非空白的部分。符号 $uq_i vB$ 表示图灵机现在的状态是 q_i , 正在扫描 v 的第一个符号,而且带上 uvB 右边的部分全部都是空白。

图灵机的格局表示能够用来跟踪图灵机的计算。符号 $uq_i vB \vdash xq_j yB$ 表示了格局 $xq_j yB$ 能够通过将 $uq_i vB$ 经过一个单独的对 M 的变换来获得。通过遵守标准的约定, $uq_i vB \vdash xq_j yB$ 表明了 $xq_j yB$ 能够通过 $uq_i vB$ 经过有限个(包括零个)单独变换来获得。当没有任何歧义的时候,该变换中代表图灵机的 M 将会被忽略。

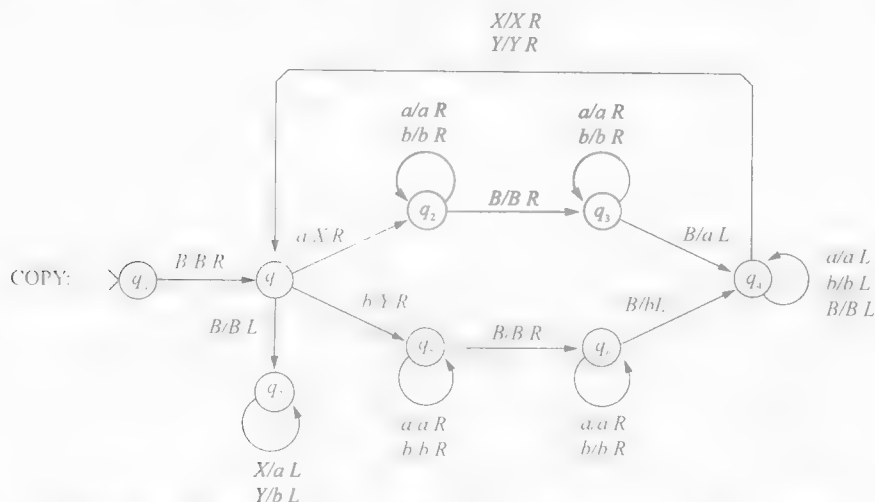
在例 8.1.1 中的图灵机将输入字符串中的 a 和 b 互相交换。当输入字符串是 $abab$ 的时候,计算的具体过程如下:

$q_0 BababB$
 $\vdash Bq_1 ababB$
 $\vdash Bbq_1 babB$
 $\vdash Bbaq_1 abB$
 $\vdash Bbabq_1 bB$
 $\vdash Bbabaq_1 B$
 $\vdash Bbabq_2 aB$
 $\vdash Bbaq_2 baB$
 $\vdash Bbq_2 abaB$
 $\vdash Bq_2 babaB$
 $\vdash q_2 BbabaB$

例 8.1.1 中的图灵机一共扫描了输入字符串两遍。第一遍是从左到右,在这个过程中交换了 a 和 b ,第二遍从右到左,仅仅将带头返回到了最左边的带位置。下一个例子主要给大家展示图灵机是如何通过计算来拷贝一个字符串的。拷贝数据的能力在很多的算法过程中十分重要。当需要拷贝的时候,该图灵机能够根据特定问题的数据类型来进行针对性的修改。

例 8.1.2 名为 COPY 的图灵机,输入字母表是 $\{a, b\}$, 将会生成输入字符串的拷贝。也就是说,带上面的字符串为 BuB 的图灵机,在执行了拷贝计算之后,带上面的字符串将会是 $BuBuB$ 。

257



计算过程中,从带的最左边输入符号开始,一次拷贝一个符号。带上的符号 X 以及 Y 则记录了输入字符串已经被拷贝的部分。输入字符串中的第一个没有标记的符号指出弧是从状态 q_1 开始。而环 q_1, q_1, q_1, q_1, q_1 则将出现出现的 a 用 X 替换,并且在生成的字符串中增加一个 a 。与此类似,下边的一个分支,则拷贝了一个 b ,并且在输入字符串中用 Y 将 b 替换。当整个字符串被拷贝完了之后,由状态 q_6 开始的转换则将 X 和 Y 分别再替换回 a 和 b ,并且将带头返回到初始的位置。□

8.2 作为语言接收器的图灵机

图灵机已经成为有效计算的范例。图灵机的计算主要是由一系列的基本计算组成,这些基本计算的决策因素包括机器现在的状态,以及带头正在读取的符号。在前一节中构造的图灵机主要用来描述图灵机计算方面的种种特性,在这些计算中需要读取以及操作带上的符号,但是并没有关于计算结果的任何解释。除了计算函数以外,图灵机同样能被设计成一个接收语言的机器。计算的结果可以根据图灵机结束的状态或者带在计算结束时的格局来定义。

本节我们将图灵机看成一个语言接收器,也就是是否接收或者拒绝某一输入字符串的计算。初始的时候,是否接收输入字符串是根据计算的最终状态来定义的。这与有限状态自动机以及下推自动机接收字符串的方式类似。但是与有限状态自动机以及下推自动机的不同之处在于,图灵机并不需要全部读完输入字符串就能够判断是否可以接收输入字符串。增加了终结状态的图灵机是一个六元组 $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 其中 $F \subseteq Q$ 是终结状态的集合。

定义 8.2.1 图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, 输入字符串 $u \in \Sigma^*$, 如果 M 计算停止在终结状态,则说此字符串被终结状态接收(accepted by final state)。如果一个计算是非正常结束,则不管机器是否停止在终结状态,此字符串都被拒绝。 M 所能够接收的语言,称为 $L(M)$, 用来表示 M 所能够接收所有的字符串。

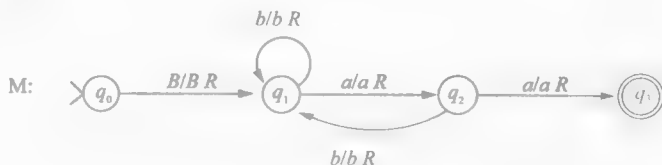
被图灵机接收的语言称为递归可枚举语言(recursively enumerable language)。图灵机能够向两个方向移动带头以及其能够处理空白的能力使得它不会因为某个特定的输入而停机,因而图灵机的计算结果有三种可能性:停机并接收输入字符串,停机并拒绝输入字符串,或永不停机。基于最后一个可能性来说,若图灵机 M 接收了 L ,则可以说其识别了语言 L ,但并不一定因为输入的字符串而停机。图灵机 M 计算并且从而识别语言 L ,但对于不属于 L 的字符串却没有任何的答案。

某种语言被图灵机所接收,但是对于其中任何的字符串,图灵机最后都会停机,则说这种语言是递归的。可以判定某个字符串是否是这种语言的成员;图灵机在计算后是否停机提供了检测一个字符串是否属于该语言的方法。这种类型的图灵机有时被说成是能够决定语言的图灵机。可递归只是语言的一个属性,而不是能够识别该语言的图灵机的属性。某一种语言,可能会有多个图灵机能够接收,

有些可能对于任何输入均会停机,而有一些却不一定。但如果存在一个能够对于所有的输入字符串均停机的图灵机,便是以说明可以判定字符串是否属于某种语言,并且该语言是一种递归语言。

在第12章我们将展示一些能够被图灵机所识别,但是却不能够被判定的语言。因而断定,递归的语言是递归可枚举语言的真子集。术语递归以及递归可枚举来源于图灵机可计算性的功能性解释,这点将在第13章详细介绍。

例 8.2.1 图灵机 M



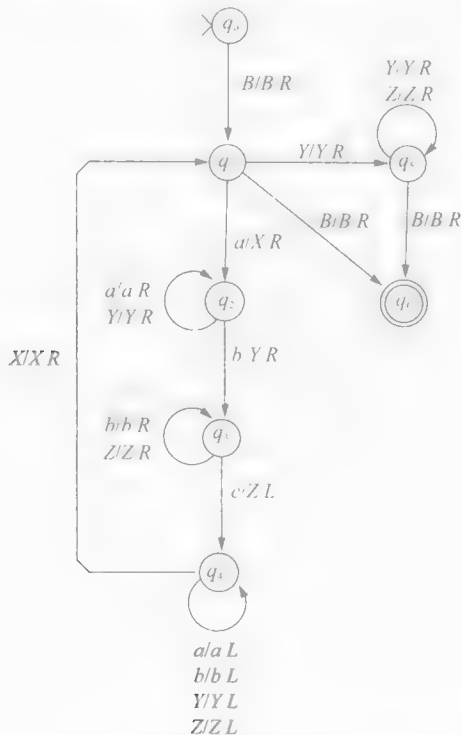
[260]

接收语言 $(a \cup b)^* aa(a \cup b)^*$, 下面的计算

$$\begin{aligned} & q_0 BaabbB \\ \vdash & Bq_1 aabbB \\ \vdash & Baq_2 abbB \\ \vdash & Baaq_3 bbB \end{aligned}$$

在接收字符串 $aabb$ 之前仅仅检查了前半段。语言 $(a \cup b)^* aa(a \cup b)^*$ 是递归的, 图灵机 M 将会接收每一个输入的字符串然后停机。当遇到子串 aa 的时候, 计算成功地结束了。其他的计算则会在读输入字符串右边第一个空白的时候停机。□

例 8.2.2 语言 $L = \{a^i b^j c^k \mid i \geq 0\}$ 被图灵机 (下图) 所接收:



带符号 X 、 Y 和 Z 来标记 a 、 b 、和 c 从而来判断他们是否匹配。当输入的字符串中的符号全部被正确地转成了带符号的时候, 整个计算便正确地结束了。从状态 q_1 到 q_6 的变换也接收空字符串。

[261]

图灵机 M 显示了 L 是递归的。在识别 L 中的字符串时, 计算停止在 M 为 q_0 的状态。对于那些不属于 L 的字符串, 当图灵机一旦发现其不匹配 $a'b'c'$ 的模式的时候, 就会停止在不接收的状态, 比如说, 若输入字符串为 bca , 则 M 会在状态为 q_1 的时候停机; 同理, 输入字符串为 abb 的时候, M 会在状态 q_3 的时候停机。□

8.3 可供选择接收标准

通过定义 8.2.1, 图灵机是否接收字符串决定于计算停止时的图灵机状态。在这部分中, 提出了定义接收的一些可供选择的方法。

第一个能够做的选择是通过停机来定义接收。在这样的图灵机中, 若输入某字符串, 计算结束的时候图灵机停机了, 这就说明该字符串被图灵机接收了。而使得机器非正常停机的计算则说明字符串不被图灵机接收。当通过停机来确定接收的时候, 图灵机的定义是一个五元组 $(Q, \Sigma, \Gamma, \delta, q_0)$, 在这里, 终结状态对于确定该图灵机接收的语言没有任何作用, 从而被忽略了。

定义 8.3.1 假设图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0)$, 若图灵机 M 对于输入字符串 $u \in \Sigma^*$, 计算能够正常终止, 则说明该字符串被停机方式接收 (accepted by halting)。

被设计成为停机方式接收的图灵机主要用于语言的识别。对于任何不属于该语言的字符串的计算将不会终止。定义 8.3.2 说明了任何能够被停机方式接收的语言, 同样能够被终结状态接收的图灵机所识别。

定理 8.3.2 下面的叙述是等价的:

- i) L 被终结状态接收的图灵机所接收。
- ii) L 被停机方式接收的图灵机所接收。

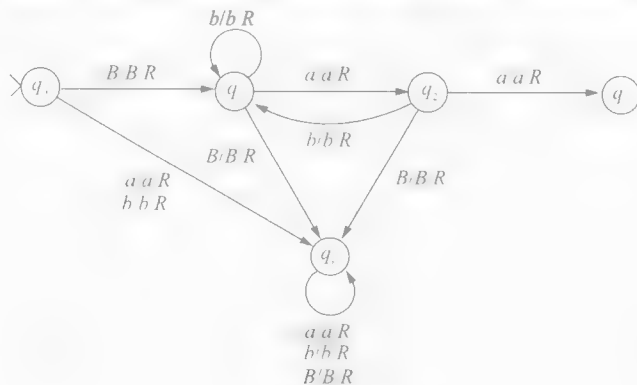
证明: 若图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0)$ 停机接收语言 L , 则图灵机 $M' = (Q, \Sigma, \Gamma, \delta, q_0, Q)$, 其中每一个状态都是终止状态, 一定能够终结状态接收语言 L 。

反之, 若图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 终结状态接收语言 L , 定义图灵机 $M' = (Q \cup \{q_r\}, \Sigma, \Gamma, \delta', q_0)$ 停机接收为:

- i) 若 $\delta(q_i, x)$ 已经被定义, 则 $\delta'(q_i, x) = \delta(q_i, x)$ 。
- ii) 对于每一个状态 $q_i \in Q - F$, 若 $\delta(q_i, x)$ 未被定义, 则 $\delta'(q_i, x) = [q_r, x, R]$ 。
- iii) 对于每一个 $x \in \Gamma$, $\delta'(q_r, x) = [q_r, x, R]$ 。

在图灵机 M 中和 M' 中, 接收字符串的计算是完全一样的。图灵机 M 中不成功的计算可能终止在拒绝状态, 不正常终止或者不能停止。若图灵机 M 中的不成功计算终止, 那么图灵机 M' 中的计算则会进入状态 q_r 。进入状态 q_r 后, 图灵机将会不确定地向右移动。图灵机 M' 中仅有的停机计算就是那些在图灵机 M 中在接收状态停机的计算。因而 $L(M) = L(M')$ 。■

例 8.3.1 将例 8.2.1 中的图灵机更改成为通过停机接收 $(a \cup b)^* aa(a \cup b)^*$ 。下图所示的图灵机就是根据定理 8.3.2 构造的。当整个字符串中没有遇到 aa 的时候, 计算就会进入状态 q_r 。



通过删除从 q_0 到 q_i 、 q_c 到 q_r ，以及标记为 a/aR 和 b/bR 的弧，得到的图灵机仍然停机接收 $(a \cup b)^+ aa(a \cup b)^+$ 。□

在练习 7 中，将会介绍一种称为被进入方式接收 (acceptance by entering) 的接收方式。在这种方式中，图灵机使用了终结状态，但是并不需要终结接收运算。若计算中曾经到达了终结状态，则该字符串就会被接收，而进入终结状态之后，剩下的计算则与是否接收该字符串无关。与停机接收一样，任何被进入接收的图灵机都能够被变形为接收同一语言且以终结状态接收的图灵机。

除非另有说明，图灵机都将会如定义 8.2.1 一样，采用被终结状态接收。其他设计用来接收同一种语言的图灵机，与标准图灵机是等价的。

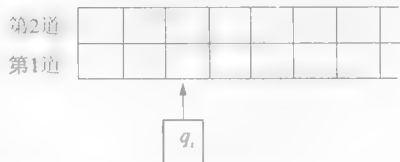
8.4 多道图灵机

本章的余下部分主要是来介绍标准图灵机模型的一些变体。每一种变体都从某种程度上增强了标准图灵机的性能。我们证明了变体图灵机所能接收的语言与标准图灵机能够接收的语言是完全一致的。另外还有的一些图灵机的变体将会出现在练习题中。

263

多道带是将带分成多条轨迹的带。多道带中的一个位置包含了带字母表中的 n 个符号。下图描述了一个两道的带，并且带头正在扫描第二个位置。

图灵机读取整个带的位置。当要选择正确的转换时，多道图灵机增加了需要考虑的信息量。两道图灵机中的一个带位置用有序对 $[x, y]$ 来表示，其中 x 是第一道的符号，而 y 是第二道的符号。



两道图灵机的状态、输入字母表、带字母表以及初始状态和终止状态与标准图灵机均没有什么区别。两道图灵机的一次

状态转换需要读取整个带位置，然后重写整个带位置。两道图灵机的一个状态转换可以这样表示： $\delta(q_i, [x, y]) = [q_j, [z, w], d]$ ，其中 $d \in \{L, R\}$ 。

两道图灵机的输入字符串被第一道中的标准输入位置所代替，第二道中的所有位置被初始化成空白。多道图灵机中使用的是被终结状态接收。

定理 8.4.1 语言 L 能够被两道图灵机接收当且仅当该语言 L 能够被标准图灵机接收。

证明：显而易见，如果语言 L 能够被一个标准图灵机所接收，那么它一定能够被一个两道图灵机所接收。等价的两道图灵机只需要简单地忽视第二道的存在即可。

设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个两道图灵机。可以构造一个单道图灵机，使得其每个带位置中记录的信息能够和两道图灵机的某个带位置中记录的信息相一致。我们只需要将两道图灵机中任意带位置的信息以有序对来表示即可完成上述需求。等价的单道图灵机 M' 的带字母表包含了两道图灵机 M 的带信息的有序对。两道图灵机的输入由第二项为空的有序对组成。 M 的输入符号 a 与 M' 的有序对 $[a, B]$ 是相同的。接收 $L(M)$ 的单道图灵机为

$$M' = (Q, \Sigma \times \{B\}, \Gamma \times \Gamma, \delta', q_0, F)$$

264

其转换方程为

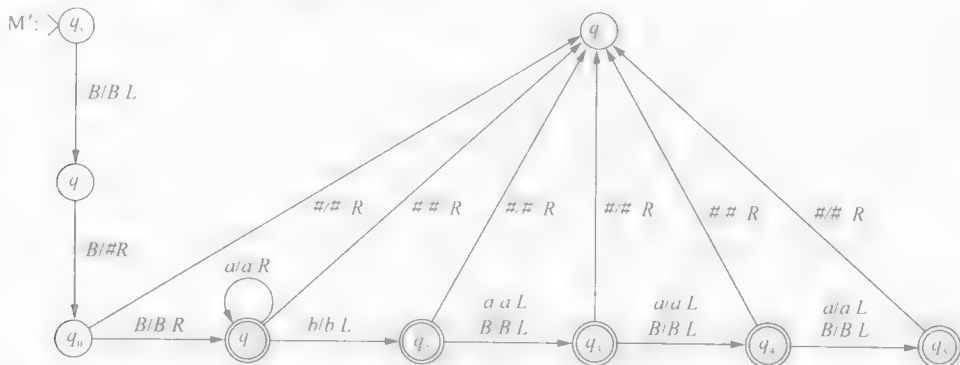
$$\delta'(q_i, [x, y]) = \delta(q_i, [x, y])。$$

8.5 双向图灵机

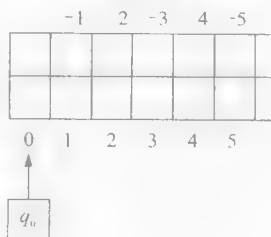
具有双向带的图灵机和标准图灵机的模型是完全等同的，不同之处仅在于双向图灵机可以向两个方向无限延伸。由于双向图灵机没有左边界，所以输入字符串可以放到带的任意位置，而其余的带位置均被假定为空白。带头在初始化的时候紧挨着输入字符串的左端。双向图灵机的主要好处在于，图灵机的设计者根本不用考虑万一越过带左边界可能会产生的麻烦。

具有双向带的双向图灵机能够通过向输入字符串的左端增加一个特殊字符来代表单向带的左边界，从而能够构造并模拟标准图灵机的动作。符号 $\#$ ，一般不在图灵机的带字母表中出现，往往用来模拟带的边界。与标准图灵机等价的双向图灵机在计算刚开始的时候，就在紧挨着带头初始位置的左

我们现在来说明被有双向带的图灵机接收的语言也能够被标准图灵机接收。这个证明使用了定理 8.4.1, 此定理建立了双向图灵机与标准图灵机之间的可互相定义的性质。双向带的带位置能够使用整数的全集来进行编号, 带头的初始位置被标号为 0, 输入字符串则是从位置 1 开始。



设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是有着双向带的图灵机。通过使用上述的双向带与两道带之间的对应, 我们能够构造一个可以接收 $L(M)$ 的单向、两道的图灵机 M' 。 M 中的状态转换被当前的状态以及当前扫描的字符所限定。而 M' , 则要扫描两道带, 并在每一个带位置读取两个符号。 M' 的状态中还包含符号 U (上面的) 以及 D (下面的) 从而指出到底是哪一道用于 M' 状态的转换。



M' 是由图灵机 M 以及符号 U 以及 D 来构造的:

$$Q' = (Q \cup \{q_i, q_f\}) \times \{U, D\}$$

$$\Sigma' = \Sigma$$

$$\Gamma' = \Gamma \cup \{\#\}$$

$$F' = \{[q_i, U], [q_i, D] \mid q_i \in F\}$$

M' 的初始状态是 $[q_i, D]$ 。从这个状态进行的转换是在最左边的带位置的上一道写入一个标记#

从 $[q_i, D]$ 开始的状态转换将带头返回到初始的位置, 从而开始对图灵机 M 的模拟。在余下的计算部分, 第二道的符号#用来指示当带头在读位置0并且状态从 U 变到 D 的时候的改变。 M' 的状态转换的定义如下

1. $\delta'([q_i, D], [B, B]) = [[q_i, D], [B, \#], R]$
2. 对于每一个 $x \in \Gamma$, $\delta'([q_i, D], [x, B]) = [[q_0, D], [x, B], L]$
3. 对于每一个 $z \in \Gamma - \{\#\}$ 以及 $d \in \{L, R\}$, 只要 $\delta(q_i, x) = [q_j, y, d]$ 是图灵机 M 的一个状态转换, 那么 $\delta'([q_i, D], [x, z]) = [[q_j, D], [y, z], d]$ 。
4. 对于每一个 $x \in \Gamma - \{\#\}$ 以及 $d \in \{L, R\}$, 只要 $\delta(q_i, x) = [q_j, y, d]$ 是图灵机 M 的一个状态转换, 那么 $\delta'([q_i, U], [z, x]) = [[q_j, U], [z, y], d']$, 其中 d' 是 d 的相反方向。
5. 只要 $\delta(q_i, x) = [q_j, y, L]$ 是图灵机 M 的状态转换, 那么 $\delta'([q_i, D], [x, \#]) = [q_j, U], [y, \#], R]$ 。
6. 只要 $\delta(q_i, x) = [q_j, y, R]$ 是图灵机 M 的状态转换, 那么 $\delta'([q_i, D], [x, \#]) = [q_j, D], [y, \#], R]$ 。
7. 只要 $\delta(q_i, x) = [q_j, y, R]$ 是图灵机 M 的状态转换, 那么 $\delta'([q_i, U], [x, \#]) = [q_j, D], [y, \#], R]$ 。
8. 只要 $\delta(q_i, x) = [q_j, y, L]$ 是图灵机 M 的状态转换, 那么 $\delta'([q_i, U], [x, \#]) = [q_j, U], [y, \#], R]$ 。

第3条所产生的状态转换模拟了图灵机 M 在带头的开始与结束位置均为正整数的状态转换情况。在模拟中, 其方法主要是通过操作带的下面一道。第4条中定义的状态转换仅仅使用了两道带的上面一道, 这与图灵机 M 中双向无穷带中负数所在位置的状态转换相对应。

剩下的状态转换则模拟了图灵机 M 双向带上从位置0开始的其他的状态转换。如果忽略状态中的符号 U 以及符号 D , 那么从带位置0开始的状态转换由第1道上的符号来确定。如果描述道的符号为 D , 那么状态转换由第5条以及第6条来定义。第7条和第8条中的定义主要应用在当前的状态为 $[q_i, U]$ 的情况。

上述非正式的描述给出了双向图灵机以及单向图灵机是等同的证据。

[267]

定理 8.5.1 语言 L 被具有双向带的图灵机所接收, 当且仅当标准图灵机能够接收语言 L 。

8.6 多带图灵机

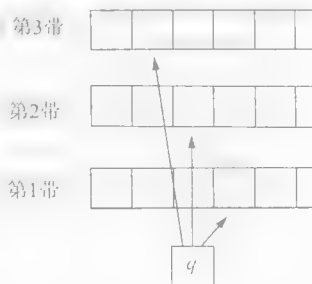
k -带图灵机有 k 个带并有 k 个独立的带头。多带图灵机的状态以及带字母表与标准图灵机的完全一致。图灵机并发地读 k 条带, 但是仅仅有一个状态。它通过连接一个控制部分和每一个独立的带头来描述当前的状态。

状态的转换由当前的状态以及每一个带头所扫描的符号来决定。

多带图灵机的状态转换有可能:

- i) 改变状态,
- ii) 在每一个带上面写一个符号,
- iii) 独立地改变每一个带头的位置。

位置的重新确定包括将当前的带头向左移动一个位置, 向右移动一个位置, 或停止在原来的位置。两带图灵机的状态转换, 首先读取1



带上的符号 x_1 以及 2 带上的符号 x_2 , 然后根据状态转换函数 $\delta(q_i, x_1, x_2) = (q_j, y_1, d_1; y_2, d_2)$ 进行状态转换, 其中 $x_1, y_i \in \Gamma$ 并且 $d_i \in \{L, R, S\}$ 。状态的转换使得图灵机在 i 带上写入符号 y_i 。符号 d_i 确切地指出了带头 i 的移动方向: L 表明向左移动, 而 R 则表明了向右移动, S 表明带头保持静止。任何带头试图移动超出带的左边界将会使得整个计算异常终止。

多带图灵机的输入字符串放在第 1 带的标准位置上, 其他所有的带上均仅有空白。带头首先扫描每个带的最左边的位置。多带图灵机的状态能够使用一个状态图来描述, 其中每一条弧上的标签详细说明了每一条带的具体动作。例如, $\delta(q_i, x_1, x_2) = (q_j, y_1, d_1; y_2, d_2)$ 将会使用一条从 q_i 到 q_j , 标记为 $[x_1/y_1, d_1; x_2/y_2, d_2]$ 的弧来表示。

268 多带图灵机的两个优势分别是: 在带与带之间拷贝数据以及比较不同带上的字符串。这两个特性均会在下面的例子中有所介绍。

例 8.6.1 图灵机



接收语言 $\{a^i b a^i \mid i \geq 0\}$ 。输入字符串为 $a^i b a^i$ 的图灵机, 在状态 q_1 的时候, 将输入字符串前面的多个 a 拷贝到带 2 上。当带 1 读取 b 的时候, 图灵机进入状态 q_2 , 然后比较带 1 上 b 之后的 a 与拷贝到带 2 上的 a 。如果两条带上的 a 一样多, 则计算终止并且接收输入字符串。如果输入字符串中没有 b , 则计算终止在 q_1 状态。如果输入字符串中有多个 b , 则计算终止在 q_2 状态。如果输入字符串中有一个 b , 但是前后的 a 的数量不等, 同样终止在 q_2 状态。由于每一个计算都能够终止, 图灵机 M 提供了一个方法来判断输入字符串是不是 $\{a^i b a^i \mid i \geq 0\}$ 的一员, 因此该语言也是递归的。

标准图灵机是只有一条带的多带图灵机。因此, 任何一个递归可枚举语言均能够被多带图灵机接收。下面, 我们将会展示两带图灵机的计算能够使用一个五带图灵机来模拟。一般地说, 任何一个能够被 k -带图灵机接收的语言, 一定能够被 $2k+1$ -道图灵机接收。多带图灵机以及标准图灵机接收语言的等价性使得我们能够做出下面的结论。□

定理 8.6.1 语言 L 被多带图灵机接收, 当且仅当标准图灵机能够接收语言 L 。

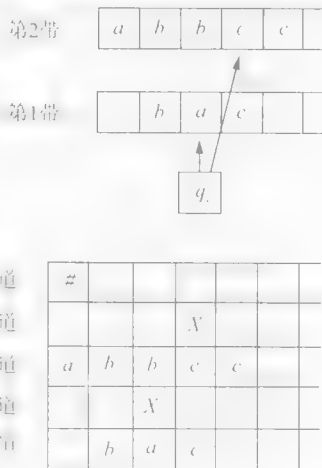
设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个两带图灵机, 在计算的过程中, 两条带的带头是独立地位于两条带之上的。

269 多道图灵机的单一带头一次读取固定位置的所有道上的数据。下面的五道图灵机 M' 就被构造用来模拟两带图灵机 M 的计算。1 道和 3 道保存了两带图灵机 1 带和 2 带上面的信息, 2 道与 4 道有一个非空的方格来指出多带图灵机的两个带头的移动方向。

多道图灵机模拟过程的第一个动作就是在第 5 道的最左边位置上写入符号 $\#$, 以及在第 2 道和第 4 道的最左边位置上写入符号 X 。多道图灵机剩下的计算过程包括了一系列的动作, 从而来模拟两带图灵机中的状态转换。

两带图灵机中的状态转换主要是由当前带头所扫描的两个符号, 以及机器当前的状态所决定的。而五道图灵机的模拟中则是将已经处理过的字符用 X 来标注, 状态则是使用形如 $[s, q_i, x_1, x_2, y_1, y_2, d_1, d_2]$ 的八元组来表示, 其中 $q_i \in Q; x_i, y_i \in \Sigma \cup \{U\}$ 且 $d_i \in \{L, R, S, U\}$ 。元素 s 代表了对 M 中状态转换模拟的状态。符号 U 被添加到带字母表以及方向集合中, 它说明了当前项为未知的。

设 $\delta(q_i, x_1, x_2) = (q_j, y_1, d_1; y_2, d_2)$ 是两带图灵机 M 的一个可用的状态转换。 M' 模拟 M 的转换的初始状态是 $[\#, q_i, U, U, U, U, U, U]$ 。下面的几个步骤在多道图灵机中模拟了两带图灵机 M 的状态转换:



1. $f1$ (发现第1个符号): M' 往右移动, 直至它读到第2道上的 X 。然后进入状态 $f1, q_1, x_1, U, U, U, U, U$, 其中 x_1 是 X 下面的第1道中的符号。当在这个状态中记录了第1道中的符号时, M' 重新返回初始位置。第5道上的符号#用来标识带头。

2. $f2$ (发现第2个符号): 同样的过程, 从而能够记录第4道的 X 符号下面的字符。 M' 进入状态 $f2, q_1, x_1, x_2, U, U, U, U, U$, 其中 x_2 是位于第3道并且恰好在 X 符号下面的字符, 然后带头再次返回到初始位置。

3. M' 进入状态 $[p1, q_j, x_1, x_2, y_1, y_2, d_1, d_2]$, 其中 q_j, y_1, y_2, d_1, d_2 均从状态转换 $\delta(q_i, x_1, x_2)$ 得来。这个状态包含了需要用来模拟 M 的信息。

4. $p1$ (打印第1个字符): M' 往右移动直到第2道的 X 符号处, 然后在第1道上面写入符号 y_1 。第2道上的符号 X 按照 d_1 所指示的方向移动, 然后机器返回到初始位置。

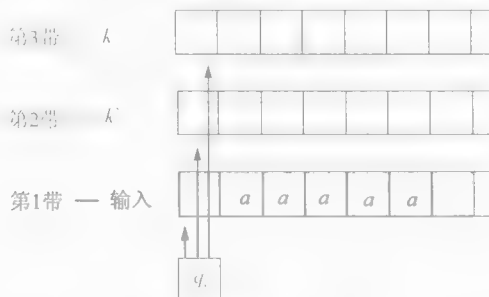
5. $p2$ (打印第2个字符): M' 往右移动直到第4道的 X 符号处, 然后在第3道上面写入符号 y_2 。第4道上的符号 X 按照 d_2 所指示的方向移动。

6. 状态转换 $\delta(q_i, x_1, x_2) = q_j, y_1, d_1, y_2, d_2$ 的模拟, 终止于将带头返回到初始位置, 然后进行下一步的状态转换。

如果 $\delta(q_i, x_1, x_2)$ 在两带图灵机中没有被定义, 那么该步骤的模拟在第2步, 即带头返回到初始的位置后停止。不管 q_i 是不是两带图灵机 M 的接收状态, 状态 $f2, q_1, x_1, y_1, U, U, U, U, U$ 均是多道图灵机 M' 的接收状态。 [270]

下面的两个例子描述了使用多条带来存储以及操作计算中的数据。

例 8.6.2 集合 $\{a^k \mid k \text{ 是一个完全平方数}\}$ 是一个递归可枚举语言。下图所示就是可接收此语言的两带图灵机。第一带容纳着输入字符串, 输入字符串将会和第二带上的字符串进行比较。第二带是由 X 所组成的字符串, 其长度是完全平方数。第三带则保存着一个字符串, 其长度是第二带上字符串长度的平方根。输入字符串为 $aaaaa$ 的计算的初始结构如下



k 以及 k^2 的值会增加, 直到第二带上字符串的长度比输入字符串长或相等。一个用来进行这种比较的图灵机包括了下面的动作:

1. 如果输入的字符串是空字符串, 则图灵机的计算将会在接收状态停止。若输入字符串不为空, 则第二带和第三带将会被初始化。在第一个位置写入符号 X , 然后三个带头移动到位置 1。

2. 第三带现在容纳着 k 个 X 的字符串, 第二带容纳着 k^2 个 X 。第一带和第二带的带头同时向右移动, 并且两个带头均扫描非空的方格。第三带的带头仍然停留在位置 1。

a) 如果两个带头同时读到一个空白, 则计算停止, 并且该字符串被接收。

b) 如果第一带的带头读到一个空白, 而第二带读到了一个 X , 则计算停止, 该字符串被拒绝。

3. 如果上面的两种情况都不发生, 则带被重新设置, 然后开始下一轮的比较。

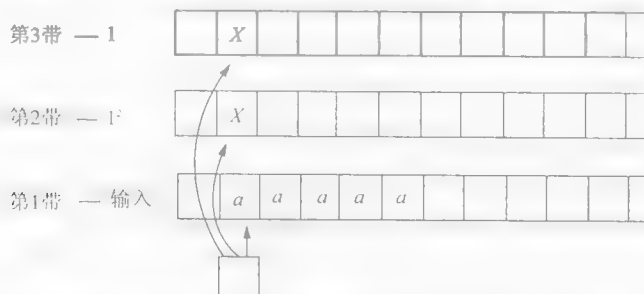
a) 在第二带的 X 字符串的右边添加一个 X 。

b) 第三带的字符串被拷贝两次并且添加到第二带。这样就建立了第二带上的 $(k+1)^2$ 个 X 字符串的序列。 [271]

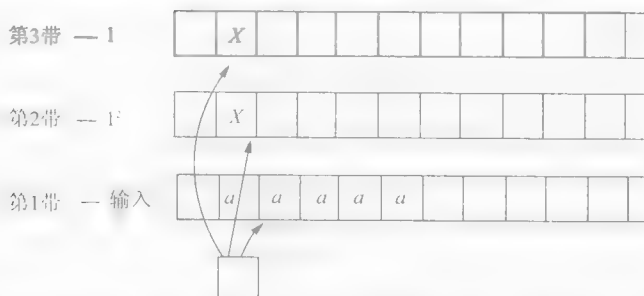
c) 在第三带的 X 符号串的右边添加一个 X ，这样就建立了第三带上面的 $k+1$ 个 X 字符的序列

4. 计算将从第二步继续往下进行。

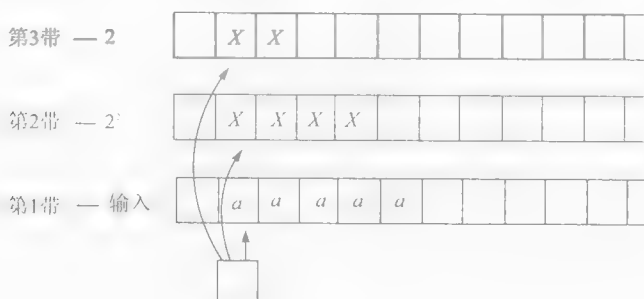
当输入字符串是 $aaaaa$ 时，我们一步一步地跟踪计算过程，第一步产生的状态如下：



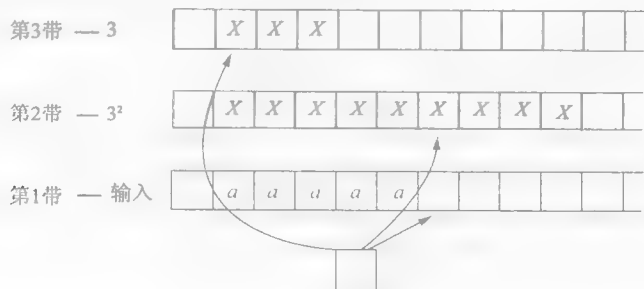
带头1以及带头2同时向右移动，当带头2在位置2遇到空白的时候，带头1和带头2停止。



[272] 第三步的(c)部分重新格式化了第二带和第三带，从而使得输入字符串能够与下一个完全平方进行比较。



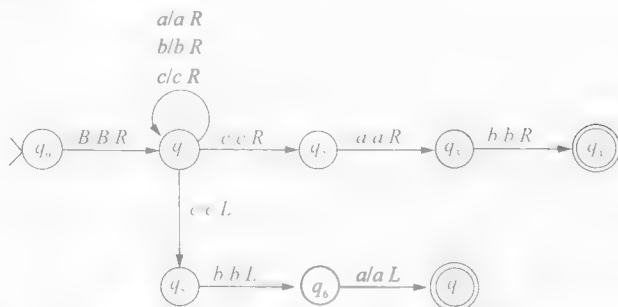
另外一个第二步的循环导致的停机并且拒绝输入字符串。



进行以上计算的图灵机可以由如下的状态转换来定义：

接收状态的计算或不能够停机的计算与此无关。和通常一样,图灵机能够接收的语言即是图灵机能够接收的字符串的集合。

例 8.7.1 下图所示的非确定型图灵机



接收包含一个 c 的字符串,且这个 c 在字符串内的位置或在 ab 之前或在 ab 之后。图灵机在状态 q_1 处理输入字符串,直到遇到 c 字符。当遇到字符 c 的时候,图灵机仍然处于 q_1 状态,然后进入 q_2 状态来确定 c 是否在 ab 前面,或者进入 q_5 来确定 c 是否在 ab 后面。在非确定型语言中,图灵机的计算会选择 c 然后选择一个条件来检查。□

在例 8.7.1 中构造的图灵机使用终结状态来接收字符串。和标准图灵机一样,非确定型图灵机的接收能够使用终结状态接收或者使用停机接收。非确定型图灵机停机接收字符串 u ,当且仅当计算的过程中有任何一个可能的计算在读字符串 u 的时候正常停止。练习 24 提出了另外一个接收同样语言的方法。

非确定型图灵机并没有增加图灵机的计算能力,被非确定型图灵机接收的语言和被确定型图灵机接收的语言并没有什么差别。为了完成非确定型图灵机到确定型图灵机之间的转化,我们将会展示只有一个输入字符串的多重计算,能够被依次产生和检查的过程。

非确定型图灵机在接收一个字符串的时候,往往会产生很多种计算过程。通过给所有可供选择的转换排序,图灵机的计算能够被系统地产生。考虑到状态的合并以及带符号的情况,我们设 n 是状态转换的最大数。计数假设:在 $\delta(q_i, x) \neq \emptyset$ 的情况下,对于每个状态 q_i 和带符号 x , $\delta(q_i, x)$ 定义了 n 个(不必不相同)转移。如果状态转换函数被定义的少于 n 个转换,那么,为了完成这种排序,某些状态转换就需要被分配多个数字。

形如 (m_1, m_2, \dots, m_k) 的序列,其中每个 i 是从 1 到 n 的数字,定义了非确定型图灵机中每一个独一无二的计算。与这个序列相关联的计算包含了 k 个或者更少的状态转换。第 j 个状态转换是由当前状态,被扫描的符号,以及 m_j 序列中的第 j 个元素共同决定的。假定第 $j-1$ 个状态使得当前图灵机的状态为正在扫描 x 的 q_i ,那么若 $\delta(q_i, x) = \emptyset$,则计算终止;否则,图灵机则会执行 $\delta(q_i, x)$ 中编号为 m_j 的转换。

表 8-1 转换的排序

状态	符号	转换	状态	符号	转换
q_0	B	$1q_1, B, R$	q_2	a	$1q_3, a, R$
		$2q_1, B, R$			$2q_3, a, R$
		$3q_1, B, R$			$3q_3, a, R$
q_1	a	$1q_1, a, R$	q_3	b	$1q_4, b, R$
		$2q_1, a, R$			$2q_4, b, R$
		$3q_1, a, R$			$3q_4, b, R$
q_1	b	$1q_1, b, R$	q_5	b	$1q_6, b, L$
		$2q_1, b, R$			$2q_6, b, L$
		$3q_1, b, R$			$3q_6, b, L$
q_1	c	$1q_1, c, R$	q_6	a	$1q_7, a, L$
		$2q_2, c, R$			$2q_7, a, L$
		$3q_5, c, L$			$3q_7, a, L$

例 8.7.1 中的非确定型图灵机的状态转换能够按照表 8-1 中一样排序, 当输入字符串为 $acab$, 且序列为 $(1,1,1,1,1)$, $(1,1,2,1,1)$ 和 $(2,2,3,3,1)$ 时, 图灵机的计算过程分别如下:

$q_0BacabB$ 1	$q_0BacabB$ 1	$q_0BacabB$ 2
$\vdash Bq_1acabB$ 1	$\vdash Bq_1acabB$ 1	$\vdash Bq_1acabB$ 2
$\vdash Baq_1cabB$ 1	$\vdash Baq_1cabB$ 2	$\vdash Baq_1cabB$ 3
$\vdash Bacq_1abB$ 1	$\vdash Bacq_2abB$ 1	$\vdash Bq_5acabB$
$\vdash Bacaq_1bB$ 1	$\vdash Bacaq_3bB$ 1	
$\vdash Bacabq_1B$	$\vdash Bacabq_4B$	

每一个转换状态右边的数字指出了为了得到下面的结构所需要的转换。第三个计算过早地结束了, 这是因为此时图灵机处于状态 q_5 且正在扫描的符号是 a , 而这个运算在图灵机中并没有定义。因为由 $(1,1,2,1,1)$ 定义的计算在状态 q_4 结束, 所以串 $acab$ 被接收。

使用上面的方法, 使得非确定型图灵机能够序列化地产生计算过程, 我们可以将任意一个非确定型图灵机转换成一个等价的确定型图灵机。设 $M = (Q, \Sigma, \Gamma, \delta, q_0)$ 是通过停机来接收字符串的非确定型图灵机, 这里我们之所以选择停机接收, 是因为这样能够降低图灵机计算的潜在输出的个数, 从而降低到了两个——图灵机停机 (接收) 或者不停机。因而在证明的过程中就只需要处理较少的情况。假定图灵机 M 已经按照前面的方法对其状态转换进行了排序, 其中转换状态的状态 (符号对) 最大数是 n 。一个确定的二带图灵机 M' 被构造成为与 M 接收同样的语言。并且同样地, M' 也是停机接收。

276

构造图灵机 M' 的目的是模拟图灵机 M 的计算过程。 M' 的序列 (m_1, \dots, m_k) 以及其计算过程之间的相关性保证了所有可能的计算都能被顾及。二带图灵机 M' 的三条带的角色分别如下:

第 1 带: 存储输入字符串;

第 2 带: 模拟 M 的带;

第 3 带: 用 (m_1, \dots, m_k) 的形式存储序列从而指导整个模拟过程的进行。

图灵机 M' 的计算过程包含下面的动作:

1. 一个从 1 到 n 的整数序列 (m_1, \dots, m_k) 被写在第 3 带上。
2. 第 1 带上的输入字符串被拷贝到第 2 带上的标准输入的位置;
3. 在第 2 带上模拟第 3 带上被序列定义的 M 的计算。
4. 如果在执行第 k 个状态转换之前模拟停止了, 则图灵机 M' 的计算停止并且接收字符串。
5. 如果第 3 步中的计算并不停止, 则第 3 带上将会生成下一个序列, 并且计算继续从第二步开始进行。

整个模拟过程由第 3 带上的序列的值来指引。图 8-1 中的确定型图灵机生成了所有的由从 1 到 n 的整数组成的有限长度的序列。其中 1、2、 \dots 、 n 是独立的带符号。长度为 1 的序列是按照数字的顺序生成的。然后是长度为 2 的序列, 再然后则是长度为 3 的序列, 等等。图灵机的计算从状态 q_0 开始, 此时带头处于位置 0 处。当带头再次回到位置 0 的时候, 带存储有下一个序列的值。符号 i/i 是 $1/1, 2/2, \dots, n/n$ 的简写。

使用这些无穷尽的数字序列, 我们就构造了一个确定型图灵机 M' , 并且接收的语言恰好是 $L(M)$ 。图灵机 M' 的计算过程交织了第 3 带上序列的生成以及第 2 带上对 M 的模拟。 M' 停止当且仅当第 3 带上的序列定义的计算在 M 上停机。因为 M 和 M' 均是停机接收的。

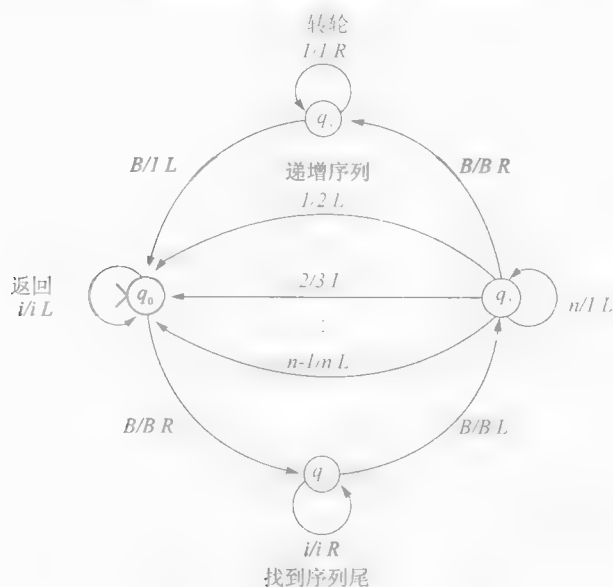
设 Σ 和 Γ 是图灵机 M 的输入字母表与带字母表。则图灵机 M' 的字母表是:

$$\Sigma_{M'} = \Sigma$$

$$\Gamma_{M'} = \{x, \#x \mid x \in \Gamma\} \cup \{1, \dots, n\}$$

$\#x$ 形式的符号代表了符号 x , 但是表示的是第 2 带在模拟图灵机 M 运算的过程中最左边的符号。图灵机 M' 的状态转换很自然地通过转换函数进行了分组。有状态的标记 q_i 用于第 3 带上序列的生成。

277

图 8-1 产生 $\{1, 2, \dots, n\}^*$ 的图灵机

这些状态转换能够由图 8-1 中描述的图灵机得到。在这个过程中，读第 1 带和第 2 带的带头保持不动。

$$\begin{aligned}
 \delta(q_{s,0}, B, B, B) &= [q_{s,1}; B, S; B, S; B, R] \\
 \delta(q_{s,1}, B, B, t) &= [q_{s,1}; B, S; B, S; i, R] & t=1, \dots, n \\
 \delta(q_{s,1}, B, B, B) &= [q_{s,2}; B, S; B, S; B, L] \\
 \delta(q_{s,2}, B, B, n) &= [q_{s,2}; B, S; B, S; 1, L] \\
 \delta(q_{s,2}, B, B, t-1) &= [q_{s,4}; B, S; B, S; t, L] & t=1, \dots, n-1 \\
 \delta(q_{s,2}, B, B, B) &= [q_{s,3}; B, S; B, S; B, R] \\
 \delta(q_{s,3}, B, B, 1) &= [q_{s,3}; B, S; B, S; 1, R] \\
 \delta(q_{s,3}, B, B, B) &= [q_{s,4}; B, S; B, S; 1, L] \\
 \delta(q_{s,4}, B, B, t) &= [q_{s,4}; B, S; B, S; t, L] & t=1, \dots, n \\
 \delta(q_{s,4}, B, B, B) &= [q_{c,0}; B, S; B, S; B, S]
 \end{aligned}$$

[278] 下一步是在第 2 带上复制输入字符串。符号 $\#B$ 被写在位置 0 上来标记带的左边界

$$\begin{aligned}
 \delta(q_{c,0}, B, B, B) &= [q_{c,1}; B, R; \#B, R; B, S] \\
 \delta(q_{c,1}, x, B, B) &= [q_{c,1}; x, R; x, R; B, S], \text{ 其中所有 } x \in \Gamma - \{B\} \\
 \delta(q_{c,1}, B, B, B) &= [q_{c,2}; B, L; B, L; B, S] \\
 \delta(q_{c,2}, x, x, B) &= [q_{c,2}; x, L; x, L; B, S], \text{ 其中所有 } x \in \Gamma \\
 \delta(q_{c,2}, B, \#B, B) &= [q_0; B, S; \#B, S; B, R]
 \end{aligned}$$

模拟图灵机 M' 第 2 带上的对图灵机 M 的计算的转换，可直接从 M 的状态转换中得到。若 $\delta(q_i, x) = [q_j, y, d]$ 是图灵机 M 中指派给排序序号为 t 的转换状态，那么

$$\begin{aligned}
 \delta(q_i, B, x, t) &= [q_j; B, S; y, d; t, R] \\
 \delta(q_i, B, \#x, t) &= [q_j; B, S; \#y, d; t, R]
 \end{aligned}$$

就是图灵机 M' 中的相应的转换状态了。

如果第 3 带上的序列包含了 k 个数字，那么顶多就模拟 k 个转换。如果图灵机 M 对于第 3 带上记载的序列停机的话，则图灵机 M' 也同样停机。如果带头读到第 3 带上面的空白，则本序列中的所有

状态转换已经被模拟完了,在下一个序列的模拟开始之前,第2带上面的模拟计算的结果都必须被全部清空。为了完成这个过程,第2带和第3带的带头都被重置到了最左边的位置,并且状态分别是 $q_{e,0}$ 和 $q_{e,1}$,然后第2带的带头向右移动,并且清空带上的符号。

$$\delta(q_i, B, x, B) = [q_{e,0}; B, S; x, S; B, S], \text{其中所有 } x \in \Gamma$$

$$\delta(q_i, B, \#x, B) = [q_{e,0}; B, S; \#x, S; B, S], \text{其中所有 } x \in \Gamma$$

$$\delta(q_{e,0}, B, x, B) = [q_{e,0}; B, S; x, L; B, S], \text{其中所有 } x \in \Gamma$$

$$\delta(q_{e,0}, B, \#x, B) = [q_{e,1}; B, S; B, S; B, L], \text{其中所有 } x \in \Gamma$$

$$\delta(q_{e,1}, B, B, t) = [q_{e,1}; B, S; B, S; t, L], t = 1, \dots, n$$

$$\delta(q_{e,1}, B, B, B) = [q_{e,2}; B, S; B, R; B, R]$$

$$\delta(q_{e,2}, B, x, i) = [q_{e,2}; B, S; B, R; i, R], \text{其中所有 } x \in \Gamma \text{ 且 } i = 1, \dots, n$$

$$\delta(q_{e,2}, B, B, B) = [q_{e,3}; B, S; B, L; B, L]$$

$$\delta(q_{e,3}, B, B, t) = [q_{e,3}; B, S; B, L; t, L] \quad t = 1, \dots, n$$

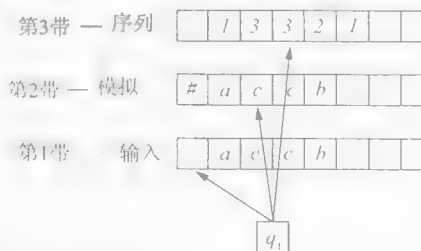
$$\delta(q_{e,3}, B, B, B) = [q_{s,0}; B, S; B, S; B, S]$$

如果第3带上面读到空白,那么带上的所有在模拟过程中读取的位置均会被清空。之后,图灵机 M' 则返回带头的初始位置并且进入 $q_{s,0}$ 状态,从而产生下一个序列并且继续对图灵机 M 的运算进行模拟。 [279]

对于图灵机 M 的计算的模拟过程,即算法的第二步到第五步,将会循环,直到第3带上某个定义停机计算的序列出现时,才会停止。对这个运算的模拟将会使图灵机 M' 停机接收输入字符串。如果输入的字符串不属于 $L(M)$,则序列的生成以及对于图灵机 M' 的计算的模拟则会不停地进行下去。

通过下面策略构造的确定型图灵机的行动使用了例 8.7.1 中的非确定型图灵机以及表 8.7.1 中的状态转换进行描述。若图灵机 M 由序列 $(1, 3, 3, 2, 1)$ 来定义,且输入字符串是 $accb$,则最开始的三个状态转换是:

$q_0BaccbB1$
 $\vdash Bq_1accbB3$
 $\vdash Baq_1ccbB3$
 $\vdash Bq_3accbB$

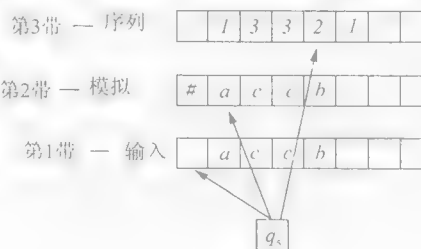


这些状态转换指出了,图灵机 M 的特定计算的序列 1, 3, 3, 2, 1 写在图灵机 M' 的第3带上。在执行 M 的第三个状态转换之前,三带图灵机 M' 的格局如右图所示。

若图灵机 M 在状态 q_1 而且正在扫描符号 c ,则转换状态 3 会导致图灵机打印出字符 c ,并且进入状态 q_3 ,带头向左移动。这个转换状态被图灵机 M' 用转换 $\delta'(q_1, B, c, 3) = [q_3; B, S; c, L; 3, R]$ 激发。根据图灵机 M 的转换状态来规定图灵机 M' 在状态转换中如何修改第二带的内容,并且移动第3带的带头,从而指出后面的状态转换的标号。

非确定型图灵机能够使用多道带、双向带甚至多带来进行模拟。使用这些可变的格局来定义的图灵机同样能够接收精确的递归可枚举语言。

和确定型图灵机一样的是,终结状态接收的非确定型图灵机能够用来判别某个语言是递归的,如果该非确定型图灵机的任何一个计算导致其终止的话,则在其等同的确定型图灵机中的计算也应该同样使图灵机终止(练习 23)。



例 8.7.2 如下所示的两带非确定型图灵机



接收由符号 a, b 组成的字符串集合, 其中 b 恰在正中间。从 q_1 到 q_2 的状态转换需要读取第 1 带上的符号 b , 这主要是用来猜测 b 是否恰在输入字符串的正中。状态 q_2 的循环比较了 b 之后跟着的符号的数目与 b 之前的符号的数目。如果输入字符串中没有包含符号 b , 则会在状态 q_1 停机, 而如果输入字符串中的 b 不在正中的话, 图灵机将会在 q_1 或 q_2 停机。因此, 既然图灵机 M 在遇到任何输入的时候均停机, 则 $L(M)$ 是一个递归语言。 \square

下一个例子显示了多带图灵机与非确定性的猜测和检查机制相结合所提供的灵活性。

例 8.7.3 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个标准图灵机, 其所接收的语言是 L 。我们下面设计一个两带非确定型图灵机 M' , 它接收字母表在 Σ 上的字符串, 这种串的一个长度大于或等于 2 的子串在语言 L 中。也就是说, $L(M') = \{u - u - xyz, \text{length}(y) \geq 2 \mid y \in L\}$ 。如果输入是 u , 则图灵机 M' 的计算包括下面几步:

1. 读取第 1 带上的输入字符串, 并且非确定地选择这个字符串中的一个位置, 从而开始拷贝子字符串到第 2 带上;
2. 从第 1 带拷贝到第 2 带上, 然后非确定地选择一个位置停止;
3. 在第 2 带上模拟图灵机 M 的运算。

开始的两个步骤包含了非确定性地猜测 u 的某一个子串, 第二步检查了是否该子串是 L 的一部分。

[281

图灵机 M' 的状态集合是 $Q \cup \{q_0, q_a, q_b, q_c, q_d, q_e\}$, 并且初始状态是 q_0 。字母表和终结状态集合与图灵机 M 是一样的。转换状态的第一步和第二步使用了状态 q_a, q_b, q_c, q_d 以及 q_e 。

$$\begin{aligned} \delta'(q_a, B, B) &= \{[q_b, B, R; B, R]\} \\ \delta'(q_b, x, B) &= \{[q_b, x, R; B, S], [q_c, x, R; x, R]\}, \text{其中所有 } x \in \Sigma \\ \delta'(q_c, x, B) &= \{[q_c, x, R; x, R], [q_d, x, R; x, R]\}, \text{其中所有 } x \in \Sigma \\ \delta'(q_d, x, B) &= \{[q_d, x, R; B, S]\}, \text{其中所有 } x \in \Sigma \\ \delta'(q_d, B, B) &= \{[q_e, B, S; B, L]\} \\ \delta'(q_e, B, x) &= \{[q_c, B, S; x, L]\}, \text{其中所有 } x \in \Sigma \\ \delta'(q_e, B, B) &= \{[q_0, B, S; B, S]\} \end{aligned}$$

从状态 q_a 到状态 q_e 的转换初始化了将 u 的子串拷贝到第 2 带上的这个过程。状态 q_e 的第二次转换完成了对于串的选择。在状态 q_d 中, 第一带的带头移动到输入字符串后面的空白处, 而第 2 带的带头则在状态 q_e 时重新移动到位置 0。

在非确定地选择了子串之后, 图灵机 M 的状态转换就在第 2 带上开始了, 同时还检查该“正在猜测的”子串是否是属于语言 L 的。这一部分计算的状态转换直接由 δ 得到, 即图灵机 M 的状态转换函数:

$\delta'(q_i, B, x) = [q_i, B, S; y, d]$, 其中 $\delta(q_i, x) = [q_i, y, d]$ 是图灵机 M 中状态转换。当图灵机在第 2 带上运算的时候, 读取第一带的带头则会一直保持静止。 \square

8.8 用来枚举语言的图灵机

在本节中的图灵机将被形式化成一个语言接收器: 图灵机被描述成具有一个输入字符串, 并且通过运算的结果来表示是否接收输入字符串。图灵机同样能够被设计用来枚举一个语言。这样设计的图灵机的运算往往依次生成语言的无穷无尽的实例。由于枚举图灵机没有任何的输入字符串, 所以该图灵机会一直计算, 直到生成该语言的任何一个实例为止。

如同接收语言的图灵机一样,有多种方式来定义一个枚举图灵机。我们在定义枚举图灵机的时候,可以使用 k -带确定型图灵机作为下面的模型,其中 $k \geq 2$ 。第一带是输出带,而其他的带则是用于工作的带。特殊的带符号#被用在输出带上,来分离在计算的过程中生成的语言集合中的每一个元素。

此处涉及的图灵机需要执行两个不同的任务,接收以及枚举。为了进行区分,接收语言的图灵机 [282] 将会被表示为图灵机 M ,而枚举图灵机则要表示成图灵机 E 。

定义 8.8.1 一个 k -带图灵机 $E = (Q, \Sigma, \Gamma, \delta, q_0)$ 枚举 (enumerate) 语言 L , 如果:

- i) 图灵机运算开始的时候,所有的带上均是空白;
- ii) 在每一个状态转换的时候,第一带(输出带)的带头或者保持不动,或者向右移动;
- iii) 在图灵机运算的任何时候,第一带上非空的部分的形式如下:

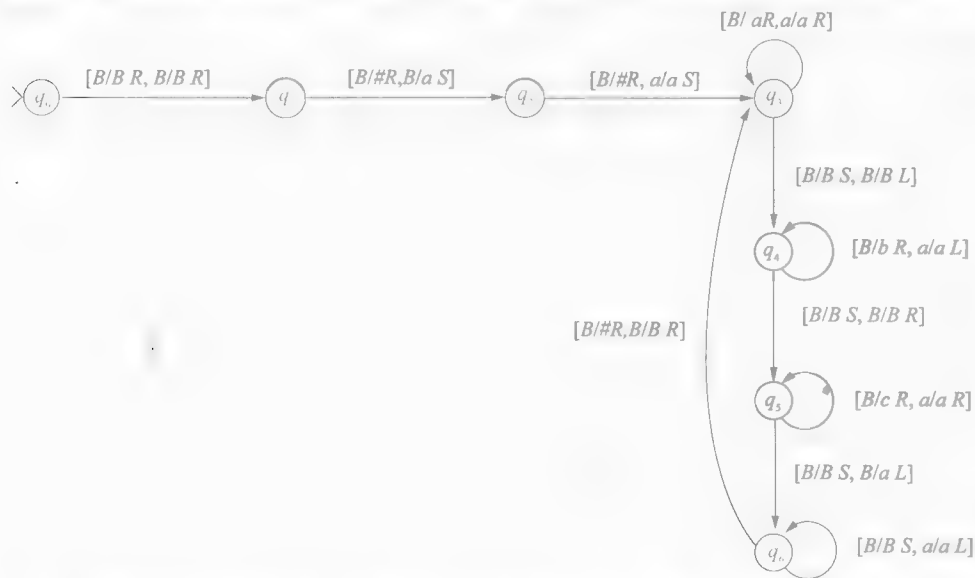
$$B\#u_1\#u_2\#\dots\#u_k\# \quad \text{或者} \quad B\#u_1\#u_2\#\dots\#u_k\#v$$

其中 $u_i \in L$ 且 $v \in \Sigma^*$

- iv) 字符串 u 将会被写在第一带上,并且前后都有一个#号,当且仅当 $u \in L$

最后一个条件表明了图灵机 E 的运算能够枚举语言 L ,并且最终将属于语言 L 的每一个字符串写到了输出带上。由于需要产生语言 L 中的每一个元素,如果图灵机 E 需要枚举的是一个无穷的语言,则图灵机永远不会停机。定义中并不要求机器停机,即使枚举的是一个有限的语言。这样的图灵机有可能在输出带上写完了语言 L 的最后一个元素之后,同样永远都不停止。

例 8.8.1 图灵机 E 枚举语言 $L = \{a^i b^j c^k \mid i \geq 0\}$ 。例 8.2.2 中给出了接收该语言的一个图灵机



图灵机 E 的计算最开始是在输出带上面写入 $\#$ 字符来表示 $\lambda \in L$ 。同时,字符 a 被写到第二带的第一个位置,且带头返回到带的第 0 个位置。在这个时候,图灵机 E 进入到如下动作所描述的不终止的循环中: [283]

1. 在工作带上面遇到任意一个符号 a 的时候,带头往右移动,并且在输出带上面写入一个符号 a 。
2. 然后工作带的带头从右向左移动,移过所有的字符 a ,且每遇到一个字符 a ,则在输入带上写入一个字符 b 。
3. 此带头向右移动,当工作带上面遇到任何一个字符 a 的时候,往输出带上写入一个字符 c 。
4. 在工作带的最后一个位置加上一个字符 a ,并且带头移动到带的第一个位置。
5. 往输出带上写入一个符号 $\#$ 。

当某一个字符串完全被写入到输出带上之后, 1 作带就包含了在这个枚举过程中需要构造的下一个字符串的信息。□

定义为枚举, 则需要该语言中的任意一个字符串均出现在输出带上, 但同时也允许一个字符串显示多次。定理 8.8.2 显示了任意一个能够被图灵机枚举的语言, 也能够被这样的图灵机枚举, 即, 该语言中的每一个字符串在输出带上仅被写一次。

定理 8.8.2 设语言 L 是图灵机 E 枚举的语言, 那么就存在能够枚举语言 L 的图灵机 E' , 并且语言 L 中的每一个字符串仅在图灵机 E' 的输出带上出现一次。

证明: 假设图灵机 E 是一个 k -带图灵机, 并且枚举语言 L 。那么, 能够通过 k -带图灵机 E 构造一个满足“单次输出”需求的 $(k+1)$ -带图灵机。直觉上, 我们觉得图灵机 E 是图灵机 E' 的一个子图灵机, E 所产生的字符串会在图灵机 E' 输出的时候考虑。在 E 的基础上再加上一条带即构成了图灵机 E' 的输出带, 而图灵机 E 中的输出带则成为了图灵机 E' 中的一条 1 作带。为了简洁, 我们把第 1 带称为图灵机 E' 的输出带。第 2 带、第 3 带、...、第 $k+1$ 带则用来模拟图灵机 E 。在模拟图灵机 E 的过程中, 第 2 带作为输出带。图灵机 E' 的动作由下面的步骤序列组成:

1. 图灵机 E' 的运算开始于第 2、3、...、 $k+1$ 带对图灵机 E 的动作的模拟。
 2. 在对图灵机 E 的模拟中, 当在第 2 带上写下 $\#u\#$ 的时候, E' 初始化一个搜索过程来检查 u 是否已经存在于第 2 带上。
 3. 如果 u 不在第 2 带上, 则将 u 添加到图灵机 E' 的输出带上。
 4. 对于图灵机 E 的模拟则重新开始, 并且开始产生下一个字符串。
- 寻找另一个 u 是否在第 2 带上, 需要带头检查第 2 带上非空白的所有部分。由于第 2 带不是图灵机 E' 的输出带, 所以输出带的带头向左移动的情况被避免了。■

定理 8.8.2 说明了使用枚举这个术语来描述这种类型的运算是正确的。一运算中属于该语言的字符串被依次无穷无尽地列举出来。生成字符串的顺序定义了自然数的初始顺序到语言 L 的映射, 因而我们能够讨论语言 L 中的长度为零的字符串、语言 L 中的第一个字符串, 等等。这种顺序与图灵机有关, 另外一个枚举图灵机可能会产生一个完全不同的顺序。

现在, 图灵机的运算能够以两种不同的方式来定义一个语言: 接收或者枚举。我们展示了这两种方法产生同样的语言。

引理 8.8.3 若语言 L 被图灵机枚举, 则 L 是递归枚举语言。

证明: 设 L 被 k -带图灵机 E 枚举。那么, 一个同样接收语言 L 的 $(k+1)$ -带图灵机 M 能够由图灵机 E 构造出来。图灵机 M 比图灵机 E 多的那条带作为输入带, 而剩余的 k 条带则是图灵机 M 用来模拟图灵机 E 的。开始的时候, 图灵机 M 的输入带上面有字符串 u , 然后图灵机 M 就开始模拟图灵机 E 的运算。当在对图灵机 E 的模拟中写下符号 $\#$ 时, 则生成字符串 $w \in L$ 。图灵机 M 则对字符串 w 和字符串 u 进行比较, 如果 $u = w$, 则接收字符串 u ; 否则, 则通过对图灵机 E 的模拟来生成语言 L 的另外一个字符串, 然后比较的过程会继续进行。若 $u \in L$, 则其终将被图灵机 E 生成, 然后被图灵机 M 所接收。■

在上述的证明中, 由于接收语言 L 的图灵机 M 并不会针对每一个输入字符串而停机, 因此这导致了枚举任何一个递归可枚举语言 L 是十分困难的。很直接地枚举语言 L 的方式是构造一个枚举图灵机来模拟图灵机 M 的运算, 从而确定某个字符串是否应该写在输出带上。这样的图灵机的动作应该是:

1. 生成一个字符串 $u \in \Sigma^*$;
2. 使用输入字符串 u 来模拟图灵机 M 的运算;
3. 如果图灵机 M 接收该字符串, 则将字符串 u 写到输出带上。
4. 继续步骤 1 直到测试完 Σ^* 中的所有字符串。

这种生成一测试的方法需要能够生成字母表 Σ 上的所有字符串的集合。这一点并不难, 具体方法后面会讲到。然而, 这种方法的第 2 步导致了它的失败。因为有可能产生一个字符串 u , 而字符串 u 可能会导致图灵机 M 永不终止。在这种情况下, 所有在字符串 u 之后的字符串都不能够产生而且同样也

不能够被测试是否属于语言 L 。

为了构造一个枚举图灵机,我们首先要介绍输入字符串的字典顺序,然后提供一个策略来保证枚举图灵机 E 将会检查字母表 Σ^* 中的每一个字符串。在非空字母表 Σ 上的字符串集合的字典顺序定义了自然数与 Σ^* 上字符串之间的一一对应。

定义 8.8.4 设 $\Sigma = \{a_1, \dots, a_n\}$ 是一个字母表, Σ^* 上的字典顺序 (lexicographical ordering) lo 被递归地定义为:

i) 基础步骤: $lo(\lambda) = 0, lo(a_i) = i$, 其中 $i = 1, 2, \dots, n$

ii) 递归步骤: $lo(a_i u) = lo(u) + i \cdot n^{\text{length}(u)}$ 。

被函数 lo 所赋的值在集合 Σ^* 上定义了一个完全的顺序。字符串 u 和 v 满足 $u < v, u = v$ 以及 $u > v$, 如果 $lo(u) < lo(v), lo(u) = lo(v)$ 以及 $lo(u) > lo(v)$ 。

例 8.8.2 设 $\Sigma = \{a, b, c\}$, 并且设 a, b 和 c 分别被赋值为 1, 2 和 3。那么字母表上的字典顺序如下:

$lo(\lambda) = 0$	$lo(a) = 1$	$lo(aa) = 4$	$lo(ba) = 7$	$lo(ca) = 10$	$lo(aaa) = 13$
	$lo(b) = 2$	$lo(ab) = 5$	$lo(bb) = 8$	$lo(cb) = 11$	$lo(aab) = 14$
	$lo(c) = 3$	$lo(ac) = 6$	$lo(bc) = 9$	$lo(cc) = 12$	$lo(aac) = 15$

□

引理 8.8.5 对于任意的字母表 Σ , 都存在一个能够以字典顺序枚举 Σ^* 的图灵机 E_Σ 。

枚举字母表 $\{0, 1\}$ 上的字符串集合的图灵机的构造留下来作为习题。

字典顺序以及吻合技术用来展示某个递归枚举语言 L 能够被图灵机枚举。设图灵机 M 是接收语言 L 的图灵机, 且图灵机 M 不需要因为任何输入字符串而停机。按照字典顺序在 Σ^* 的字符串集合上建立了一个列表 $u_0 = \lambda, u_1, u_2, u_3, \dots$, 并且建立了一个二维的图表, 其中行用 Σ^* 上的字符串来标记, 而列则使用自然数来标记。

表中的项 $[i, j]$ 被解释为“运行图灵机 M , 其中输入字符串是 u_i , 需要运行 j 步”, 使用例 1.4.2 中介绍的技术, 这个表中的有序对能够以一种对角线的方式被枚举 (练习 33)。

图灵机 E 中创建并用来枚举语言 L 的过程, 插入了图灵机 M 的运算中对有序对的枚举。而图灵机 E 的运算则是一个包含了下面步骤的循环:

1. 生成一个有序对 $[i, j]$ 。

2. 使用输入字符串 u_i , 并且进行 j 步, 或直到停机, 来进行对图灵机 M 的模拟。

3. 如果图灵机 M 接收, 则将 u_i 写到输出带上。

4. 继续步骤 1。

若 $u_i \in L$, 则图灵机 M 在其输入字符串是 u_i 的时候, 停机并且在 k 个状态转换之后接收字符串, 因而当有序对 $[i, k]$ 被处理的时候, u_i 会被写到输出带上。有序对 $[i, j]$ 的第二个元素保证了对于图灵机 M 的模拟将会在 j 步后停止。而随后发生的, 就是所有不能够停止的运算都不允许出现, 并且 Σ^* 中的每一个字符串均会被检查到。

将引理 8.8.3 与前面的论据合并就产生了下面的定理 8.8.6。

定理 8.8.6 一个语言是递归可枚举的当且仅当它能够被一个图灵机所枚举。

接收递归可枚举语言的图灵机停机并且接收语言中的每一个字符串, 但是在这个过程中, 停机并不是必须的, 尤其是当输入的字符串并不属于这个语言的时候。语言 L 是一个递归语言, 当它被对所有输入字符串均停机的图灵机所接收的时候, 由于每一个计算均停机, 因此这样的图灵机等于是提供一个决定过程, 通过这个过程来确定某一个字符串是不是属于语言 L 。递归语言的集合也能用枚举图灵机来定义。

3	$[\lambda, 3]$	$[u, 3]$	$[u_2, 3]$...
2	$[\lambda, 2]$	$[u, 2]$	$[u_2, 2]$...
1	$[\lambda, 1]$	$[u, 1]$	$[u_2, 1]$...
0	$[\lambda, 0]$	$[u, 0]$	$[u_2, 0]$...
	λ	u	u_2	...

[285]

[286]

枚举图灵机的定义并没有对语言中字符串生成的顺序做出任何的约束。以预定的计算顺序生成所需要的字符串则会提供一些用来判断其是否属于该语言的额外信息。从直觉上来看,用来确定某个字符串 u 是否属于该语言的策略,是使用枚举图灵机,并将字符串 u 和图灵机生成的每一个字符串相比较。最终 u 或者在输出带上(被接收),或者某个按序排在 u 后面的字符串被生成。由于输出字符串按序生成的,因此 u 就被忽略了且不属于该语言。于是我们就能够来确定某个字符串是否属于该语言,并且该语言是否为递归的。定理 8.8.7 展示了枚举语言有可能被冠以下面的特性,即该语言的元素能够按照顺序被枚举。

287 | 定理 8.8.7 语言 L 是递归的,当且仅当语言 L 能够以字典顺序被枚举。

证明:我们首先来证明每一个递归语言能够以字典顺序被枚举。设语言 L 是字母表 Σ 上的递归语言,而且语言 L 能够被某个图灵机 M 所接收,该图灵机 M 对所有的输入字符串都停机。能够以字典顺序枚举语言 L 的图灵机 E ,能够由图灵机 M 和图灵机 E_Σ 来构造,其中图灵机 E_Σ 能够以字典顺序枚举语言 Σ^* 。图灵机 E 是一个混合的,包含了图灵机 M 和图灵机 E_Σ 的运算。图灵机 E 的运算包括下面的循环:

1. 图灵机 E_Σ 运行,并生成字符串 $u \in \Sigma^*$ 。
2. 图灵机 M 使用 u 作为输入字符串来运行。
3. 如果图灵机 M 接收字符串 u ,则 u 即被写到图灵机 E 的输出带上。
4. 这种生成—测试的循环从步骤 1 开始继续。

由于图灵机 M 针对任意的输入均停机,所以图灵机 E 不可能在步骤 2 中不停机。因而任意的字符串 $u \in \Sigma^*$ 都将被生成且被测试其是否属于语言 L 。

现在我们就能够证明任意能够以字典顺序枚举的语言 L 都是递归的。证明将依据语言 L 的集合的势被分成两个不同的部分。

情形 1:语言 L 是有限的。因为任何有限的语言都是递归的,所以可知语言 L 是递归的。

情形 2:语言 L 是无限的。这个情形与定理 8.8.2 中给出的类似,只有一点除外,就是定理 8.8.2 中的顺序数是用来终止图灵机的运算的。和前面的一样,一个 $(k+1)$ -带的接收语言 L 的图灵机 M 能够由一个以字典顺序枚举语言 L 的 k -带图灵机构造出来。图灵机 M 的额外一条带用来作为输入带,图灵机 M 剩下的 k 条带允许其模拟图灵机 E 的运算。图灵机 E 产生的字符串的顺序提供了当输入的字符串不属于语言 L 时,图灵机 M 停机所需要的信息。图灵机 M 的运算开始于字符串 u 在输入带上,下面就是图灵机 M 模拟图灵机 E 的运算了。当在模拟的过程中生成了字符串 w 时,图灵机 M 会将 u 和 w 进行比较,如果 $u = w$,则图灵机 M 停机并且接收字符串 u ;如果 w 在顺序上比 u 大,则图灵机 M 停机并且拒绝该输入字符串。最后,如果 w 在顺序上比 u 小,则图灵机 E 的模拟过程重新开始,以便生成语言 L 的另外一个元素,然后继续进行这种比较的过程。 ■

8.9 练习

1. 设图灵机 M 由下表定义

δ	B	a	b	c
q_0	q_1, B, R			
q_1	q_2, B, L	q_1, a, R	q_1, c, R	q_1, c, R
q_2		q_2, c, L		q_2, b, L

- a) 当输入字符串是 $aabca$ 的时候,具体描述图灵机 M 的运算过程。
- b) 当输入字符串是 $bcbcb$ 的时候,具体描述图灵机 M 的运算过程。
- c) 给出图灵机 M 的状态图。
- d) 描述图灵机 M 的某个计算过程的结果。

2. 设图灵机 M 由下表定义

δ	B	a	b	c
q_0	q_1, B, R			
q_1	q_1, B, R	q_1, a, R	q_1, b, R	q_2, c, L
q_2		q_2, b, L	q_2, a, L	

- 当输入字符串是 $abcb$ 的时候, 具体描述图灵机 M 的运算过程。
 - 当输入字符串是 $abab$ 的时候, 具体描述图灵机 M 的前六个状态转换的运算过程。
 - 给出图灵机 M 的状态图。
 - 描述图灵机 M 的某个计算过程的结果。
- 当输入字母表是 $\{a, b\}$ 的时候, 构造一个图灵机来完成下面的每一个操作。要注意的是, 在状态 q_i 的时候, 无论计算是否终止, 带头都正在扫描第 i 个位置。
 - 将输入字符串向右移动一个位置。输入格局是 q_0BuB , 结果格局是 q_1BBuB 。
 - 将输入字符串翻转之后再连接到输入字符串后边。输入格局是 q_0BuB , 结果格局是 $q_1BBuu^R B$ 。
 - 将输入字符串的每两个字符中加入一个空白, 例如: 输入格局是 q_0BabaB , 结果格局是 $q_1BaBbBaB$ 。
 - 将输入字符串中的 b 全部删除。例如: 输入格局是 $q_0BbabaababB$, 结果格局是 $q_1BaaaaB$ 。
 - 当输入字母表是 $\{a, b, c\}$ 的时候, 构造一个图灵机, 其接收的字符串要求第一个字符 c 需要紧跟在子串 aaa 之后, 图灵机只能够接收含有一个字符 c 的字符串。
 - 当输入字母表是 $\{a, b\}$ 的时候, 构造一个图灵机, 并且要使用终结状态方式来接收下面的每一个语言。
 - $\{a^i b^j \mid i \geq 0, j \geq i\}$
 - $\{a^i b^j a^i b^j \mid i, j > 0\}$
 - 具有相同数目的 a 和 b 的字符串
 - $\{uu^R \mid u \in \{a, b\}^*\}$
 - $\{uu \mid u \in \{a, b\}^*\}$
 - 修改 5 (a) 中的图灵机, 使得该图灵机能够使用停机方式来接收语言 $\{a^i b^j \mid i \geq 0, j \geq i\}$ 。
 - 另外一种使用终结状态方式接收的图灵机定义如下: 字符串 u 被图灵机 M 接收, 如果输入字符串是 u 的图灵机 M 的计算进入终止状态 (不一定会停机)。在这个定义下, 即使图灵机不停机, 字符串也能够被其接收。证明这种定义下的被接收的语言是递归可枚举语言。
 - 单带确定型图灵机的状态转换能够使用一个部分函数来定义, 函数的定义域是 $Q \times \Gamma$, 值域是 $Q \times \Gamma \times \{L, R, S\}$, 其中 S 指出带头保持不动。证明以这种方式定义的图灵机接收的语言是递归可枚举语言。
 - 原子图灵机 (atomic turing machine) 是这样的图灵机, 每一个转换均包括一个状态的改变以及一个其他的动作, 其中的动作可能是在带上面写入字符或者移动带头, 但是不能同时做这两件事情。证明原子图灵机接收的语言是递归可枚举语言。
 - 上下文有关的图灵机 (context-sensitive turing machine) 是这样的图灵机, 决定其状态转换的因素不仅仅是当前正在扫描的字符, 还与带头右边的带方块中的符号有关, 其状态转换有如下的形式:

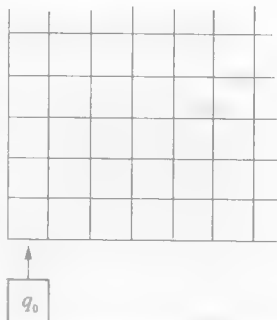
$$\delta(q_i, xy) = [q_j, z, d] \quad x, y, z \in \Gamma; d \in \{L, R\}$$
 当图灵机现在处于状态 q_i , 且正在扫描字符 x 时, 上述的状态转换仅当带头右边的带位置中恰好有一个 y 的时候才能够发生。在这种情况下, 使用 z 来替换 x , 然后图灵机进入状态 q_j , 带头向 d 指出的方向移动一格。
 - 设图灵机 M 是一个标准图灵机。定义一个上下文有关的图灵机 M' 来接收语言 $L(M)$ 。提示: 根据图灵机 M 的状态转换来定义图灵机 M' 的状态转换。
 - 设 $\delta(q_i, xy) = [q_j, z, d]$ 是一个上下文有关的状态转换。证明这个状态转换应用的结果能够通过标准图灵机的一系列的状态转换来得到。你必须考虑两种情况, 即 $\delta(q_i, xy)$ 什么时候是可用的以及不可用的。
 - 使用上述两点来得出结论: 上下文有关的图灵机接收的语言是递归可枚举语言。
 - 证明每一个递归可枚举语言都能够被仅有一个接收状态的图灵机接收。

12. 构造一个双向图灵机, 其输入字母表是 $\{a\}$, 当带包含非空白符号的时候停机. 符号 a 可能在带上的任意一个位置, 而不一定在紧挨着带头的右边方块中。

290

13. 二维图灵机 (two-dimensional turing machine) 是这样的一个图灵机, 其带包含了二维数组的带方块。

状态的转换包括了重写某个方块, 以及将带头移动到相邻的四个方块中的任意一个中。图灵机的运算开始于带头在最角落的方块。这个二维图灵机的状态转换被写作 $\delta(q_i, x) = [q_j, y, d]$, 其中 d 可能是 U (上)、 D (下)、 L (左) 以及 R (右) 这四个方向中的任意一个。设计这样的一个二维图灵机, 其输入字母表是 $\{a\}$, 而且当带包含非空方块的时候停机。



14. 设 L 是字母表 $\{a, b\}$ 上的回文的集合:

a) 设计一个接收语言 L 的标准图灵机;

b) 设计一个接收语言 L 的两带图灵机, 其中当输入字符串是 u 的时候, 图灵机的运算不会超过 $3length(u) + 4$ 个状态转换。

15. 当输入字母表是 $\{a, b\}$ 的时候, 构造一个接收语言 $a^i b^{2i} \mid i \geq 0$ 的两带图灵机, 其中, 输入带的带头仅仅从右向左移动。
16. 当输入字母表是 $\{a, b, c\}$ 的时候, 构造一个接收语言 $a^i b^j c^k \mid i \geq 0$ 的两带图灵机。
17. 当输入字母表是 $\{a, b\}$ 的时候, 构造一个接收具有相同 a 和 b 的字符串的两带图灵机。当输入字符串是 u 的时候, 图灵机的运算不会超过 $2length(u) + 3$ 个状态转换。
18. 构造一个两带图灵机, 其接收的字符串中, 每一个 a 后面 b 的个数是递增的, 也就是说, 字符串需要具有下面的形式:

$$ab^{n_1} ab^{n_2} \cdots ab^{n_k}, k > 0$$

其中 $n_1 < n_2 < \cdots < n_k$ 。

19. 构造一个非确定型图灵机, 其输入字母表是 $\{a, b\}$, 并且输入字符串包含一个满足下面两个属性的子串 u :

i) u 的长度大于或者等于 3;

ii) u 包含的符号 a 和符号 b 一样的多。

291

20. 构造一个接收语言 $L = \{uvw \mid u \in \{a, b\}^*, v, w \in \{a, b\}^*\}$ 的两带非确定型图灵机, 如果一个字符串包含两个不重叠的, 完全一致的, 长度为 5 的子串的话, 则说明该字符串是语言 L 的一个元素。当输入字符串是 w 的时候, 图灵机需要在至多 $2length(w) + 2$ 个状态转换之后停止。

21. 构造一个接收语言 $L = \{uu \mid u \in \{a, b\}^*\}$ 的两带非确定型图灵机。当输入字符串是 w 的时候, 图灵机需要在至多 $2length(w) + 2$ 个状态转换之后停止。如果是例 8.6.2 中定义的确定型图灵机, 其接收语言 L , 在输入字符串的长度是 n 的情况下, 最大的状态转换的次数是多少?

22. 设图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个标准图灵机, 它接收语言 L 。设计一个接收 Σ^* 上字符串 w 的图灵机 M' , 当且仅当 w 有一个子串是语言 L 中的元素。

23. 设语言 L 是非确定型图灵机接收的语言, 其中每一个计算都能够结束, 证明语言 L 是递归的。

24. 如果假设条件换成非确定型图灵机, 证明定理 8.3.2。

25. 证明每一个有限语言都是递归的。

26. 证明语言 L 是递归的当且仅当 L 与 L 的补集都是递归可枚举的。

27. 证明递归语言在进行并、交以及补运算时是封闭的。

28. 图 8-1 展示了产生所有由整数 1 到 n 组成的序列的图灵机, 当 $n=3$ 时, 试写下最开始的七个循环。一个循环包括带头再次回到初始位置并且又进入 q_0 状态。

29. 构造一个图灵机, 该图灵机枚举所有 $\{a\}$ 上的偶数长度的字符串。

30. 构造一个枚举集合 $\{a^i b^j \mid 0 \leq i \leq j\}$ 的图灵机。

31. 构造一个枚举集合 $\{a^{2^n} \mid n \geq 0\}$ 的图灵机。

32. 构造一个枚举 Σ^* 的图灵机 E_Σ ，其中 $\Sigma = \{1, 0\}$ ，注意：该图灵机有可能需要考虑枚举所有有限长的比特串。
- *33. 构造一个图灵机，该图灵机枚举有序对 $\mathbf{N} \times \mathbf{N}$ ，使用 $n+1$ 个 1 的串来代表数字 n 。有序对 $[i, j]$ 的输出如下：代表数字 i 的串后面紧跟着一个空白，然后是代表数字 j 的串，而标记符 # 则需要包围着整个有序对。
34. 在定理 8.8.7 中，关于每一个递归语言都能够以字典顺序枚举的证明，将有限和无穷语言分开考虑了，对于无穷语言的证明也许对于有限语言来说并不充分，为什么？
35. 定义两道非确定型图灵机的组成。证明这些图灵机接收的语言恰是递归可枚举语言。
36. 证明每一个上下文无关的语言都是递归的。提示：构造一个两带的非确定型图灵机来模拟下推自动机的运算过程。 [292]

参考文献注释

图灵机最初作为算法计算的一个模型由图灵 [1936] 提出。图灵最初设计的图灵机是确定型的，包含一个双向无限带以及一个带头。无独有偶，波斯特 [1936] 提出了一系列的抽象机器，且这些机器和图灵机具有同样的计算能力。

使用图灵机来进行函数的运算将在第 9 章详细讨论，而图灵机作为语言的识别者时的能力以及限制则会在第 10 章和第 11 章里介绍。Kleene [1952]、Minsky [1967]、Brainerd、Landweber [1974] 以及 Hennie [1977] 著的书中对图灵机的计算能力给出了一个全面的介绍。

[293]
[294]

第9章 图灵可计算函数

在前面的章节中,图灵机为接收语言提供了一个可计算的框架。运算的结果由终结状态或者停机来确定。在任意情况下均有两种不同的结果:接收或者拒绝。图灵机的计算结果同样可以在停止时,通过在带上写下某些符号来定义。通过停机时带中所允许的格局的无限多种可能性来定义最终的结果。在这种方式下,图灵机的计算产生了从输入字符串到输出字符串之间的一个映射,也就是说,图灵机计算了一个函数。当字符串用自然数来解释时,图灵机则能够用来计算数论函数。下面,我们将展示出几种重要的数论函数是图灵可计算的,并且这种可计算性在函数的组合之下也是相近的。而在第13章我们则会对图灵机所能计算的函数集合进行一个更加全面的分类。

本章的结束部分会总结怎样使用图灵机的体系结构来定义高级编程语言。这一点将图灵机与现代生活中我们所熟悉的可计算范型——计算机紧密地联系在了一起。

9.1 函数的计算

[295] 函数 $f: X \rightarrow Y$ 是一种映射,对定义域 X 中的每一个元素,最多在值域 Y 中分配一个值与之相对应。从计算的角度来说,我们将 f 的变量作为函数的输入。函数的定义并不明确地指出怎样从输入 x 如何获取 $f(x)$ ——通过函数 f 指派给 x 的值,但是却可以设计相应的图灵机来计算函数的值。图灵机计算出的某个函数的定义域和值域,包含了由图灵机字母表生成的字符串。

能够计算函数的图灵机有两个特殊的状态:初始状态 q_0 和终止状态 q_f ,计算开始于由初始状态开始的一个转换,此时图灵机的带头停留在输入字符串的开始位置。自此以后,图灵机不会再次进入 q_0 状态,该状态所仅有的作用就是初始化图灵机的计算;而所有会终止的计算都会在状态 q_f 终止,从带的第一个位置开始,将函数值写到带上。这些条件都在定义 9.1.1 中被形式化了。

定义 9.1.1 确定型单带图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ 计算一元函数 $f: \Sigma^* \rightarrow \Sigma^*$, 如果:

- i) 从状态 q_0 开始仅有一个转换,且这个转换形式如下: $\delta(q_0, B) = [q_i, B, R]$;
- ii) 不存在如下形式的转换: 对于任意的 $q_i \in Q, x, y \in \Gamma$ 以及 $d \in \{L, R\}, \delta(q_i, x) = [q_0, y, d]$;
- iii) 不存在如下形式的转换: $\delta(q_f, B)$;
- iv) 输入字符串为 u 时,只要 $f(u) = v$,那么计算在格局为 $dfBvB$ 的时候停机;而且
- v) 只要 $f(u) \uparrow$,计算将会永远进行下去。

如果存在一个能够计算某一个函数的图灵机,则说该函数是图灵可计算的 (turing computable)。当然,计算函数 f 的图灵机,在输入字符串为 u 的时候也有可能失败,在这种情况下,函数 f 则对于 u 是没有定义的。因而图灵机能够计算全函数以及部分函数。

任意的一个函数不一定非要有相同的定义域和值域。图灵机能够设计成为计算从 Σ^* 到某一个特定的集合 R 的函数,其中输入字母表是 Σ ,值域是 R 。条件 (iv) 可以被解释为要求字符串 v 是 R 的一个元素。

为了强调图灵机的特殊状态 q_0 和 q_f ,能够计算函数的图灵机应该由下面类似的图进行表示:

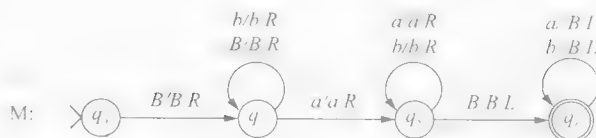
从直觉上来说,计算将一直在标记为 M 的方框中进行计算,

直到终止。这个图从某种意义上来说有些过于简单了,因为定义 9.1.1 允许有多个状态转移到状态 q_f 以及从 q_f 转移出去。然而,条



件 (iii) 则确定了当图灵机在扫描一个空白的时候,没有任何转移状态是由 q_f 开始的。当这种情况发生的时候,计算就会终止,并且将结果写到带上。

例 9.1.1 图灵机



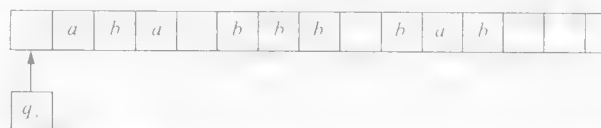
计算了从 $\{a, b\}^*$ 到 $\{a, b\}^*$ 的部分函数 f , 定义如下:

$$f(u) = \begin{cases} \lambda & (\text{如果 } u \text{ 包含一个 } a) \\ \uparrow & (\text{其他情况}) \end{cases}$$

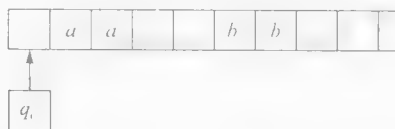
如果输入字符串中不包含某个符号 a 的话, 那么该函数 f 则称为未定义的。在这种情况下, 图灵机在状态为 q_1 的时候无限不确定地向右移动。当遇到符号 a 的时候, 图灵机则会进入 q_2 状态, 然后继续读取输入字符串的剩下部分。当将带上的输入字符串全部擦除并返回到初始位置的时候, 图灵机的计算便完成了。图灵机产生格局 $q_1 B B$ 并结束, 从而指出了将空字符串作为计算的结果。□

例 9.1.1 中的图灵机 M 设计用来计算一元函数 f , 当输入字符串不具备我们所期望的形式, 且图灵机 M 的计算不满足定义 9.1.1 的需求的时候, 我们既不用吃惊, 也不用惊慌。当输入字符串是 $BbBbBaB$ 时, 图灵机 M 结束时的格局是 $BbBbq_1 B$ 。在这个停机的格局中, 带上并没有包含一个值, 而且带头并不在正确的位置上。这正是经历了时间考验的计算机科学规律“垃圾进, 垃圾出”的另一个体现。

对于多个参数的函数的计算也很类似。输入字符串被放置在磁带上, 而参数则使用空白来分开。某三元函数 f , 输入字符串为 aba 、 bbb 以及 bab 时, 其计算的初始化如下:



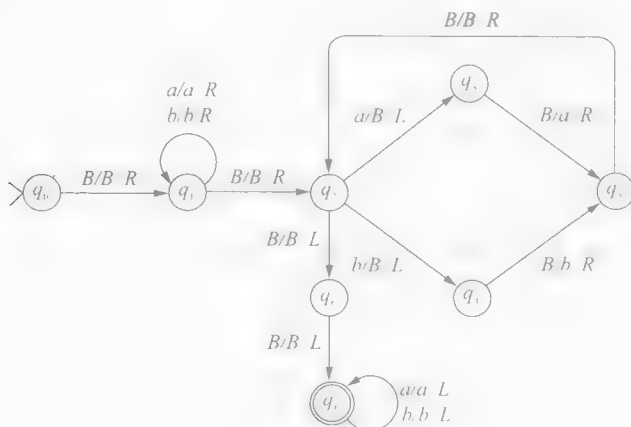
如果 $f(aba, bbb, bab)$ 被定义, 则计算终止的格局是 $q_1 B f(aba, bbb, bab) B$ 。而函数 $f(aa, \lambda, bb)$ 的计算的初始格局是



带上连续的第 3 和第 4 个位置上的空白说明了第二个参数是空字符串。

[297]

例 9.1.2 图灵机



计算了一个主要用来连接字母表 a, b 上面的字符串的二元函数。图灵机计算的初始格局是 $q_0 B u v B$ ，其中输入字符串是 u 和 v ，且两个输入字符串都不是空字符串。

初始的字符串在状态 q_1 被读入，由状态 q_2, q_3, q_4 和 q_5 形成的环将符号 a 向左移动一位，同样地，由 q_2, q_3, q_4 和 q_5 形成的环将符号 b 往左移动一位。这些环一直循环地运行直到整个第二个参数往左移动一位，从而产生格局 $q_7 B u v B$ 。□

计算函数的图灵机同样能够用来接收语言。语言 L 的特征函数 (characteristic function) 是这样的函数 $\chi_L: \Sigma^* \rightarrow \{0, 1\}$ ，定义如下：

$$\chi_L(u) = \begin{cases} 1, & \text{若 } u \in L \\ 0, & \text{若 } u \notin L \end{cases}$$

如果存在一个图灵机 M 能够计算 χ_L ，则语言 L 是一个递归语言。图灵机 M 的计算结果表明了是否接收输入字符串。而图灵机计算如下的部分特征函数

$$\hat{\chi}_L(u) = \begin{cases} 1, & \text{若 } u \in L \\ 0 \text{ 或者 } \uparrow, & \text{若 } u \notin L \end{cases}$$

则表明了语言 L 是递归可枚举的。练习 2、练习 3 以及练习 4 提出了图灵机的接收语言以及计算语言的特征函数之间的等价性。

9.2 数值计算

我们可以看到图灵机能够用于计算函数的值，且这些函数的定义域和值域都是根据带字母表生成的字符串。在本节中，我们将把注意力转移到数字计算上来，尤其是一些数论函数的计算。数论函数是有着这样的形式的函数 $f: \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \rightarrow \mathbf{N}$ 。定义域包含 n 元组的自然数。由 $sq(n) = n^2$ 定义的函数 $sq: \mathbf{N} \rightarrow \mathbf{N}$ 是一个一元的数论函数。标准的加操作与乘操作则都是二元的数论函数。

从符号计算到数值计算的转变只需要更换一个视角，因为数字是由符号组成的字符串来代表的。图灵机的输入字母表是由计算中使用到的自然数的代表来确定的。我们使用自然数 n 来代表字符串 1^n ，数值 0 使用字符串 1 来代表，而数值 1 则使用字符串 11 来表示，依此类推。这种符号模式被认为是自然数的一元表示 (unary representation)。自然数 n 的一元表示被记为 n 。当数字使用这种一元表示来编码的时候，计算数论函数的图灵机的输入字符串是只有一个元素的集合 $\{1\}$ 。

计算三元数论函数 $f(2, 0, 3)$ 的图灵机的初始机器格局是



如果 $f(2, 0, 3) = 4$ ，则计算结束时的格局是

一个 k 个变量的全数论函数 $r: \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \rightarrow \{0, 1\}$ 定义了其定义域上的一个 k 元的关系 R ，这个关系如下定义：

$$[n_1, n_2, \dots, n_k] \in R \text{ (若 } r(n_1, n_2, \dots, n_k) = 1)$$

$$[n_1, n_2, \dots, n_k] \notin R \text{ (若 } r(n_1, n_2, \dots, n_k) = 0)$$

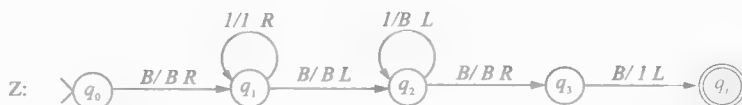


函数 r 则被称作关系 R 的特征函数 (characteristic function)。若某一个关系的特征函数是图灵可计算的，则该关系也被称作是图灵可计算的。

现在我们要构造几个图灵机来计算几个简单但是很重要的数论函数。函数使用小写字母来表示，而相应的图灵机则使用大写字母来表示。后继函数为 $s(n) = n + 1$



零函数为 $z(n) = 0$



空函数为 $e(n) \uparrow$



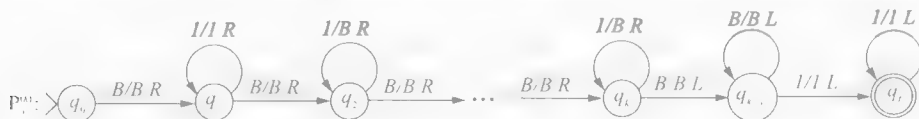
图灵机在计算后继函数的时候仅仅需要在输入字符串的右边加上一个1。零函数的计算则是擦除输入字符串并在带的第一个位置写上字符1。空函数则是对于任何参数均是没有定义的，图灵机在状态 q_1 时无穷往右移动。

零函数也能够由下图所示的图灵机来计算。



这两个计算同一个函数的图灵机描述了函数以及算法之间的区别。函数是定义域中的元素到值域中的元素进行的一个映射。只要函数被定义好了，图灵机就会机械地计算函数的值。它们之间的区别只不过一个是定义，另一个是计算而已。在9.5节中，我们将会看到有些数论函数不能够使用图灵机来计算。

k 个变量映射函数 $p_i^{(k)}$ 的值被定义为输入字符串的第 i 个参数，即 $p_i^{(k)}(n_1, n_2, \dots, n_i, \dots, n_k) = n_i$ 。上标 k 表示参数的个数，而下标则指出了用来定义映射结果的参数。上标 k 用括号括起来的原因是为了避免被认为是一个指数。而图灵机计算 $p_1^{(k)}$ 则只是将第一个参数留下，而将剩下的参数全部擦除。



300 |

函数 $p_1^{(1)}$ 仅仅将一个输入字符串映射到其自身。这个函数也被称作恒等函数 (identity function) 并被记作 id 。计算 $p_i^{(k)}$ 的图灵机 $P_i^{(k)}$ 将在例9.3.1中提及。

例9.2.1 图灵机 A 计算自然数相加的二元函数。

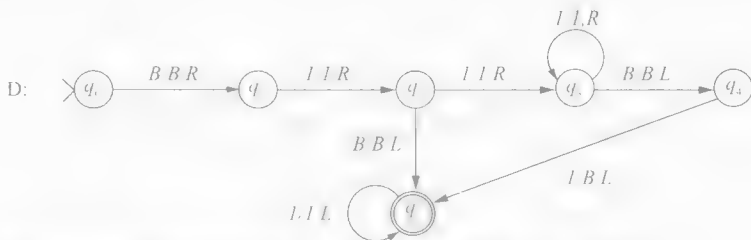


对于自然数 m 和 n ，它们的一元表示分别为 1^{m+1} 和 1^{n+1} 。它们的和由 1^{m+n+1} 来表示，现将两个参数中间的空白用一个1来替代，然后在第二个参数的右边擦除两个1，就生成了结果字符串。□

例9.2.2 先驱函数

$$\text{pred}(n) = \begin{cases} 0, & \text{若 } n=0 \\ n-1, & \text{其他情形} \end{cases}$$

由图灵机 D 来计算



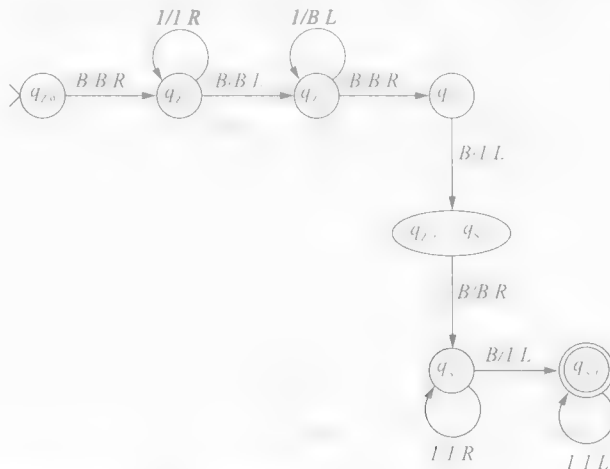
如果输入的自然数大于0，则图灵机的运算过程中会将带的最右边的1擦除

□

9.3 图灵机的顺序操作

完成单个任务的图灵机能够被合并起来，从而构造一个能完成复杂任务的图灵机。从直观上来说，这种合并往往是通过将图灵机顺序运行而得到的，即一个图灵机计算的结果成为下一个图灵机计算的输入。计算常数函数 $c(n) = 1$ 的图灵机能够通过将计算零函数的图灵机与计算后继函数的图灵机合并起来得到。无论什么输入，当图灵机 Z 的计算终止时，带上面的值是 0，然后根据这个带上的值，使用图灵机 S 进行计算，就能够产生数字 1 了。

如果图灵机 Z 的带头在位置零，并且正在扫描一个空白，那么此时 Z 终止；这恰恰是图灵机 S 的输入条件。定义 9.1.1 中介绍的初始条件以及终止条件正好可以用于推进这类计算机器之间的结合。这种机器之间信息的传递是靠将图灵机 Z 的终止状态标志成为图灵机 S 的初始状态来完成的。除了他们之间的信息传递，这两个图灵机的状态都是被认为有区别的。可以将每个状态原来所在的机器名字记为状态的下标从而来保证每个状态之间的区别。



这两个图灵机的顺序组装能够用下面的图表来表示：



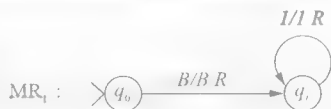
从初始状态到终止状态的节点的状态名均被省略了，这主要是因为它们都来自以前所在的子图灵机。

在图灵机的计算中，经常会遇到一些特定的执行顺序。图灵机能够被构造来执行这些重现的任务。这些图灵机被设计成能够作为在一些极为复杂的图灵机中运行的构件的形式。借一个汇编语言的术语来说，我们把那种仅执行单一任务的图灵机称为宏（macro）图灵机。

宏图灵机的运算遵循定义 9.1.1 中的几个约束。初始状态 q_0 严格地用来初始化整个计算。由于这些宏图灵机能被组合成更为复杂的图灵机，所以我们并不假定在每次运算时带头都一定都在第 0 个位置，但是我们可以假定图灵机以扫描一个空白来表示图灵机开始一个新的运算。根据操作，带上的紧临

带头左边或者正右边的部分将会在计算中被扫描到。一个宏图灵机将会包含几个状态, 其中一个运算可能会终止。与能够计算函数的图灵机相比, 宏图灵机是不允许存在从终止状态 q_i 开始且形如 $\delta(q_i, B)$ 的状态转换的。

同种类型的宏图灵机往往被描述成为一个模式。宏图灵机 MR_i 表示将带头往右移动经过 i 个连续的自然数 (1 的连续序列)。宏图灵机 MR_1 如右图所示。 MR_k 则通过添加状态使得带头往右移动连续 k 个自然数。



这种移动的宏图灵机并不影响带头所在初始位置左边的带。宏图灵机 MR_i 的运算的初始状态格局是 $B \bar{n}_1 q_0 B \bar{n}_2 B \bar{n}_3 B \bar{n}_4 B$, 而终止状态的格局是 $B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 q_i B \bar{n}_4 B$ 。

宏图灵机, 除了输入字符串有一个特殊的形式以外, 其他都与计算函数的图灵机一样。一向右移动的宏图灵机 MR_i 需要在图灵机的初始状态的时候, 带上带头右边的部分至少要有 i 个自然数的序列。这种组装图灵机的设计要求每一个宏图灵机都必须有恰好的输入格局。

几类宏图灵机通过描述图灵机计算的结果来定义。每一个宏图灵机的运算在带上的表示就是初始以及终止状态的空白符号。该宏图灵机的运算将不会访问或者更改两端的限度之外的任何数据。带头所在的位置将会用下划线表示, 而双向的箭头表示了运算前和运算后相同的带位置。

ML_k (向左移动):

$$\begin{array}{c} B \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \quad k \geq 0 \\ \updownarrow \quad \updownarrow \\ \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \end{array}$$

[303]

FR (发现右边)

$$\begin{array}{c} B B' \bar{n} B \quad i \geq 0 \\ \updownarrow \quad \updownarrow \\ B' B \bar{n} B \end{array}$$

FL (发现左边)

$$\begin{array}{c} B \bar{n} B' B \quad i \geq 0 \\ \updownarrow \quad \updownarrow \\ \underline{B} \bar{n} B' B \end{array}$$

E_k (擦除)

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \quad k \geq 1 \\ \updownarrow \quad \updownarrow \\ \underline{B} B \quad \cdots \quad B B \end{array}$$

CPY_k (复制)

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B B B \cdots B B \quad k \geq 1 \\ \up \quad \up \quad \up \\ \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \end{array}$$

CPY_{k,i} (复制 i 个数字)

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \bar{n}_{k+1} \cdots B \bar{n}_{k+i} B B \quad \cdots \quad B B \quad k \geq 1 \\ \updownarrow \quad \updownarrow \quad \updownarrow \quad \updownarrow \\ \underline{B} \bar{n}_1 B \bar{n}_2 B \cdots B \bar{n}_k B \bar{n}_{k+1} \cdots B \bar{n}_{k+i} B \bar{n}_1 B \bar{n}_2 B \quad \cdots \quad B \bar{n}_k B \end{array}$$

T (翻译)

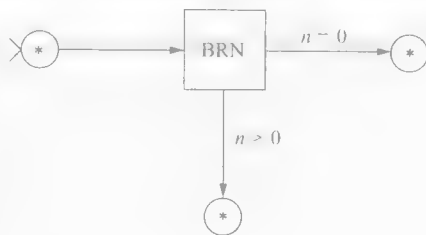
$$\begin{array}{ccc}
 BB^i \bar{n} B & i \geq 0 \\
 \updownarrow & \updownarrow \\
 B \bar{n} B^i B
 \end{array}$$

用于寻找的宏图灵机将带头往右或者往左移动到恰好能够访问第一个自然数的位置；而 F_i 则擦除了 k 个连续自然数之外，还将带头停留在其最初的位置。

[304]

用于复制的宏图灵机产生了指定的整数数字拷贝，用于产生拷贝的带段事先必须是空白 $CPY_{k,i}$ 则需要连续的 $k+i$ 个整数后面紧跟着的空白带是足够长的，以便能够产生最初的 k 个数字的拷贝。而用于翻译的宏图灵机则将第一个自然数的位置更改到带头的右边，然后当其带头在计算初始的位置时，计算终止，并且翻译好的字符串恰在带头的右边。

BRN 宏自动机有两个可能的终止状态，宏 BRN 的输入，它是一个自然数，用于选择该宏图灵机终止的状态，宏图灵机如右图所示。宏图灵机 BRN 的计算并不会改变带上的数据也不会改变带头的位置。实际上，它能够在任意形如 $B\bar{n}B$ 的格局下运行。这个宏往往用于复杂的图灵机中循环的构造以及从两者中选择一个分支的计算中。



更多的宏图灵机均能够使用这些以及预先定义的宏图灵机来构造 图灵机



交换两个数字的顺序，这个图灵机的带格局是 INT (交换)

$$\begin{array}{ccc}
 B\bar{n} B \bar{m} B B^{n+1} B \\
 \updownarrow & & \updownarrow \\
 B\bar{m} B \bar{n} B B^{n+1} B
 \end{array}$$

在练习 6 中，将会要求读者使用宏图灵机 INT 来构造一个图灵机，并且在计算的过程中不离开带片断 $BnBmB$ 。

例 9.3.1 图灵机用于评价映射函数 p_i^k 的运算，这包括三个不同的动作：擦去初始的 $i-1$ 个参数，将第 i 个参数移动到带位置 1 的地方，然后擦除输入字符串剩余的部分。用于计算 p_i^k 的图灵机可以使用宏图灵机 FR 、 FL 、 E_i 、 MR_i 以及 T 来设计。

[305]



在设计复杂图灵机的时候，用于计算函数的图灵机同样能像宏图灵机一样使用。但和宏图灵机的运算不同的是，在这样一个函数图灵机的运算中，没有前述的那些宏图灵机所需要的带上的边界。而且，这些图灵机只有在输入字符串后面全部是空白带时才能运行。

例 9.3.2 宏图灵机以及前面构造的图灵机能用来设计一个能够计算 $f(n) = 3n$ 的图灵机



使用例 9.2.1 里面的图灵机 A 进行构造，并添加了两个自然数 $f(n)$ 将完成拷贝的宏和机器 A 结合在一起，从而产生 n 的三个拷贝。输入字符串使 \bar{n} 的运算产生了下面的带格局。

如果第一个参数为0,则运算中会擦除第二个参数,并且返回到初始状态,并且停机。否则的话,图灵机 MULT 的运算就是将 m 与其自身相加 n 次。加法是先拷贝 m 然后将这个拷贝加到前面的结果中去,循环次数的记录则是当一次拷贝结束了,就将第一个参数中的1替换成 X 。□

9.4 函数的合成

如果使用函数的另外一种解释——从定义域到值域的映射——的话,可以使用下面的图表来表示一元数论函数 g 和 h 。



而从 N 到 N 的映射则可以通过确认 g 的值域以及 h 的定义域,然后更改图表中的箭头来获取。



通过这种合并获得的函数叫做函数 h 和函数 g 的合成。在定义 9.4.1 中有一元函数的合成的形式化定义。而定义 9.4.2 中则扩展到 n -元函数。

定义 9.4.1 设 g 和 h 是一元数论函数。函数 h 和 g 合成产生的函数 $f: N \rightarrow N$ 定义如下:

$$f(x) = \begin{cases} \uparrow & \text{若 } g(x) \uparrow \\ \uparrow & \text{若 } g(x) = y \text{ 且 } h(y) \uparrow \\ h(y) & \text{若 } g(x) = y \text{ 且 } h(y) \downarrow \end{cases}$$

合成的函数被记作 $f = h \circ g$ 。

如果输入是 x ,则合成函数 $f = h \circ g$ 的值被写成 $f(x) = h(g(x))$ 。值 $h(g(x))$ 有定义当且仅当 $g(x)$ 有定义且 h 对于 $g(x)$ 的值也有定义。而且,全函数的合成也会产生一个全函数。

.308

从可计算的角度来看, $h \circ g$ 的合成包含了对函数 g 和 h 的顺序上的考虑。函数 g 的计算结果为函数 h 提供了输入(如右图所示)。函数的合成有定义仅当计算的顺序使得计算能够正确的结束。

定义 9.4.2 设 g_1, g_2, \dots, g_n 是一个 k -元数论函数,设 h 是一个 n -元函数。输入论函数,而 k -元函数 f 的定义如下:

$$f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$$

它被称为 h 和 g_1, g_2, \dots, g_n 的合成 (composition), 并被写成 $f = h \circ (g_1, g_2, \dots, g_n)$ 。该函数 $f(x_1, \dots, x_k)$ 是没有被定义的, 如果

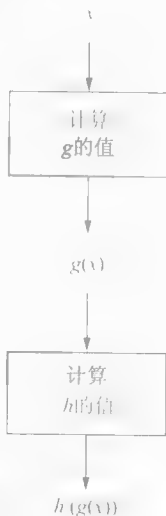
- i) 对于某些 $1 \leq i \leq n$, $g_i(x_1, \dots, x_k) \uparrow$, 或者
- ii) $g_i(x_1, \dots, x_k) = y_i$, 其中 $1 \leq i \leq n$ 且 $h(y_1, \dots, y_n) \uparrow$ 。

通常函数合成的定义也有一些可计算性的解释。输入参数被提供给每一个函数 g_i , 而这些函数则生成函数 h 的参数。

例 9.4.1 考虑如下合成函数定义的映射

$$\text{add} \circ (c_2^{(3)}, \text{add} \circ (p_1^{(3)}, p_3^{(3)})),$$

其中 $\text{add}(n, m) = n + m$, 且 $c_2^{(3)}$ 是由 $c_2^{(3)}(n_1, n_2, n_3) = 2$ 所定义的三元常数函数。此合成是一个三元函数的合成, 从最里层的函数合成开始, 那些直接需要使用输入的函数都是要有三个参数的。此函数将第一个和第三个参数相加, 结果再与第二个常数相加。输入参数是 1、0 和 3, 函数的计算结果是



309

$$\begin{aligned}
& add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)}))(1, 0, 3) \\
&= add \circ (c_2^{(3)}, (1, 0, 3), add \circ (p_1^{(3)}, p_3^{(3)})(1, 0, 3)) \\
&= add(2, add(p_1^{(3)}(1, 0, 3), p_3^{(3)}(1, 0, 3))) \\
&= add(2, add(1, 3)) \\
&= add(2, 4) \\
&= 6.
\end{aligned}$$

□

如果一个函数是通过组合图灵可计算函数来合成的话,那么其本身也是图灵可计算的。这个论点是构造性的,某个图灵机通过合并能够计算构成函数以及宏图灵机,从而被设计成为能够计算合成函数的图灵机。

设 g_1 和 g_2 是二元图灵可计算函数,并且设 h 是一个两元图灵可计算函数。由于 g_1 、 g_2 和 h 都是图灵可计算的,因此存在能够计算这些函数的图灵机 G_1 、 G_2 以及 H 。假设输入参数为 n_1 、 n_2 和 n_3 , 计算合成函数 $h \circ (g_1, g_2)$ 的过程具体如下。

机 器	格 局
	$\underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B$
CPY ₃	$\underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B$
MR ₃	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 \underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B$
G ₁	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 \underline{B} g_1(n_1, n_2, n_3) B$
ML ₃	$\underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \overline{g_1(n_1, n_2, n_3)} B$
CPY _{3,1}	$\underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \overline{g_1(n_1, n_2, n_3)} B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B$
MR ₄	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \overline{g_1(n_1, n_2, n_3)} \underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B$
G ₂	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \overline{g_1(n_1, n_2, n_3)} \underline{B} \overline{g_2(n_1, n_2, n_3)} B$
ML ₁	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 \underline{B} \overline{g_1(n_1, n_2, n_3)} B \overline{g_2(n_1, n_2, n_3)} B$
H	$B \bar{n}_1 B \bar{n}_2 B \bar{n}_3 \underline{B} h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3)) B$
ML ₃	$\underline{B} \bar{n}_1 B \bar{n}_2 B \bar{n}_3 B \overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$
E ₃	$\underline{B} B \cdots B \overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$
T	$\underline{B} \overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$

[310]

计算中首先复制了输入参数,并且使用函数 g_1 计算了这些新产生的参数拷贝的值。由于图灵机 G_1 并不会移动到初始位置的左侧,所以初始的输入保持不变。如果 $g_1(n_1, n_2, n_3)$ 是未被定义的,则图灵机 G_1 的运算会无穷进行下去。在这种情况下,整个图灵机的运算就不能停止,从而也就正确地表示 $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$ 也是未被定义的。在 G_1 正确停止的基础之上,输入再次被复制,然后图灵机 G_2 才能开始利用这些拷贝,进行下面的计算。

若 $g_1(n_1, n_2, n_3)$ 和 $g_2(n_1, n_2, n_3)$ 均为有定义的,则图灵机 G_2 也能够根据其输入正确终止,且此时带上而留下的就是图灵机 H 所需要的输入。然后图灵机 H 就开始计算 $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$ 。当图灵机 H 的计算终止后,计算结果将会被转移到正确的位置。

上述的构造过程能够很容易的产生任意元的函数的合成,从而产生了下面的定理 9.4.3。

定理 9.4.3 图灵可计算函数在合成操作上是封闭的。

定理 9.4.3 能用来解释某一个函数 f 是图灵可计算的,而并不需要清楚地构造一个能够计算该函数的图灵机。如果 f 能够用图灵可计算函数的合成来定义,那么根据定理 9.4.3, f 也是图灵可计算的。

例 9.4.2 k 元常数函数 $c_i^{(k)}$, 其值由 $c_i^{(k)}(n_1, \dots, n_k) = i$ 来定义,是图灵可计算的。函数 $c_i^{(k)}$ 可由

$$c_i^{(k)} = \underbrace{s \circ s \circ \dots \circ s \circ z \circ p_1^{(k)}}_{i \text{ times}}.$$

来定义。这个映射函数接受 k 个变量作为输入，并且将第一个参数传给零函数，然后使用 i 个后继函数的合成产生了所需要的值。由于合成的每一个函数都是图灵可计算的，因此根据定理 9.4.3，函数 $c_i^{(k)}$ 也是图灵可计算的。□

例 9.4.3 二元函数 $smSq(n, m) = n^2 + m^2$ 是图灵可计算的。这个求平方和的函数能够写成下面的函数的合成：

$$smSq = add \circ (sq \circ p_1^{(2)}, sq \circ p_2^{(2)}),$$

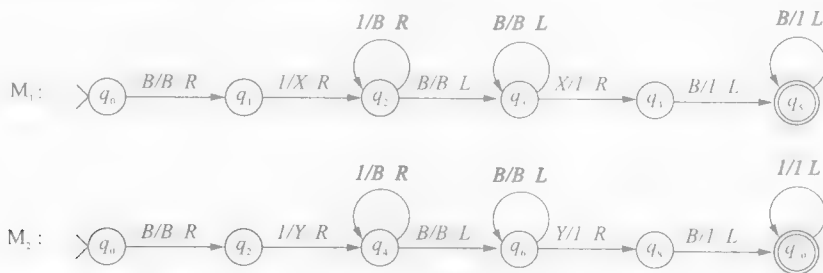
其中 sq 被定义为 $sq(n) = n^2$ 。用于加法的函数则是由例 9.2.1 中构造的图灵机来实现的，而 sq 则是由下图所示的图灵机来实现的。



9.5 不可计算函数

一个函数是图灵可计算的当且仅当存在一个可以计算该函数的图灵机。已经存在的非图灵可计算的数论函数能够使用一个简单的计数参数来描述。我们这就开始说明可计算函数的集合是可数无穷大的。

图灵机是完全由其转换函数来定义的。图灵机的状态以及计算过程中出现的带字母表均能够从状态转换中提取出来。设图灵机 M_1 和 M_2 如下定义。



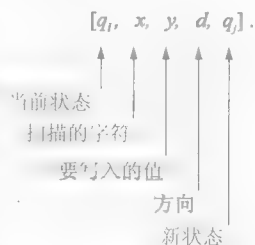
图灵机 M_1 和 M_2 都能够计算一元常数函数 $c_1^{(1)}$ 。这两个图灵机的区别仅仅是状态的名字以及在计算中标记的名字。这些符号对于计算的结果以及该图灵机能够计算的函数没有任何影响。

由于状态的名字以及带符号（除了 B 和 I ）都是非实质的，因此我们采用下面的约定来考虑图灵机中的元素的命名：

- i) 状态的集合是 $Q_0 = \{q_i \mid i \geq 0\}$ 的一个有限子集。
- ii) 输入字母表是 $\{1\}$ 。
- iii) 带字母表是集合 $\Gamma_0 = \{B, I, X_i \mid i \geq 0\}$ 的一个有限子集。
- iv) 初始的状态是 q_0 。

图灵机的状态转换使用函数表示法来标识；当状态为 q_i 且带符号是 x 时，状态转换被记为 $\delta(q_i, x) = [q_j, y, d]$ 。这些信息也能够用五元组来表示。使用如前所述的命名规则，图灵机的一个状态转换是集合 $T = Q_0 \times \Gamma_0 \times \Gamma_0 \times \{L, R\} \times Q_0$ 的一个元素。由于集合 T 是可数集合的笛卡尔积，则 T 也是可数的。

确定型图灵机的状态转换形成集合 T 的一个有限子集，其中每一个元素的最初两个部分是有区别的；这样的子集只有可数个，从而也就说明了



图灵可计算函数的个数最多就是可数无穷大的。从另一方面来说,图灵可计算函数的数目最少也是可数无穷大的,这主要是因为有着众多的常数函数,由例 9.4.2 知,它们也都是图灵可计算的。上述这些说明引出了定理 9.5.1。

定理 9.5.1 图灵可计算的数论函数的集合是可数无穷大的。

在 1.4 节中,对角化技术被用于证明存在无穷多的一元数论全函数,结合定理 9.5.1,我们可以得出推论 9.5.2。

推论 9.5.2 存在一个一元数论全函数,且它不是图灵可计算的。

推论 9.5.2 轻描淡写地描述了可计算和不可计算方程之间的关系。前者组成了一个可数的集合,而后者则组成了一个不可数的集合。

9.6 关于编程语言

高级编程语言在可计算系统中被十分广泛地运用。程序定义了机械和确定型的过程,以及算法的计算特点。直觉上看,用程序语言写并且运行在计算机中的程序,能够用图灵机来模拟,这依靠的仅是机器指令更改了内存中部分位置的部分比特而已。这恰恰是图灵机所能够执行的操作,即在内存中写 1 或者 0。虽然说需要很多次的状态转换才能够完成这个任务,但这并不难让我们想象到,连续的状态转换能够访问正确的位置并且重写内存中的内容。

313

在本节中,我们将会仔细地探讨一下,使用图灵机的体系结构作为高级语言的底层框架的可能性。基于图灵机体系结构的编程语言的发展,更深层次地描述了图灵机模型的能力。在描述汇编语言时,我们使用图灵机以及宏图灵机来定义其中的操作。本节的目的并不是创造一个功能性的汇编语言,而是描述图灵机体系结构的普遍适应性。

标准图灵机提供的可计算的框架,在本节中均有涉及。我们将会设计一种汇编语言 TM,从而为图灵机体系结构与编程语言之间建立一个跨越它们鸿沟的桥梁。这种汇编语言的第一个目标就是提供对于图灵机动作的一个顺序描述。图灵机的“程序流”是由图灵机状态图中的弧来表示的。汇编语言程序的流则由指令的顺序执行组成,除非这种模式被某个重定向流的指令特定地改变了。在汇编语言中,分支以及 goto 指令都用来改变顺序程序流,而汇编语言的第二个目标是提供简化内存管理的指令。

图灵机的底层体系结构用来评估汇编语言程序(参见图 9-1)。输入值被赋给变量 v_1, \dots, v_k , 而 v_{k+1}, \dots, v_n 则是程序中的局部变量。变量的值被顺序存储并且使用空格分开。输入变量在图灵机计算函数时位于标准输入位置。一个 TM 程序开始于声明程序中要用到的局部变量,每一个局部变量在计算开始之前都被初始化为 0。

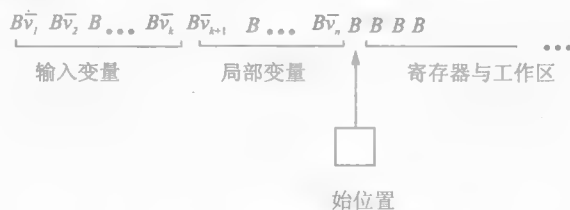


图 9-1 用于高级计算的图灵机结构

当初始化完成后,带头停在变量与带剩余空白之间的空白处,这个地方被称为中心位置(home position)。在每两条指令的运行之间,带头都会返回到中心位置。在中心位置的右边是图灵机里的寄存器。中心位置右边的第一个值被当作寄存器 1 的值,第二个值是寄存器 2 的值,依此类推。寄存器的值也必须被顺序赋值,也就是说,第 i 个寄存器能够被读或写当且仅当寄存器 1, 2, \dots , $i-1$ 已经被分配了相应的值。在表 9-1 中给出了汇编语言 TM 的指令。

314

表 9-1 TM 指令

TM 指令	解 释
INIT v_i	把局部变量 v_i 初始化为 0
HOME $_t$	当 t 变量被分配后, 将带头移动到中心位置
LOAD v_i, t	将变量 v_i 的值装载到寄存器 t
STOR v_i, t	将寄存器 t 中的值存储到 v_i 的地址
RETURN v_i	删除变量, 并把变量 v_i 的值保留在输出位置
CLEAR $_t$	删除寄存器 t 中的值
BRNL, t	如果寄存器 t 的值是 0 则转移到标记为 L 的指令
GOTOL	执行标记为 L 的指令
NOP	无操作 (与 GOTO 命令结合使用)
INC $_t$	增加寄存器 t 的值
DEC $_t$	减小寄存器 t 的值
ZERO $_t$	用 0 替换寄存器 t 中的值

带初始化的完成使用了 INIT 以及 HOME 命令。INIT v_i 保存了局部变量 v_i 的位置, 并且将其赋值为 0。由于变量在带上面被顺序存储, 则本地变量必须在 TM 程序初始的时候被顺序初始化。当本地变量的初始化完成之后, HOME 指令即将带头移动到中心位置, 这些指令的定义如右表所示。

指令	定义
INIT v_i	MR_{i-1} ZR ML_{i-1}
HOME $_t$	MR_t

[315]

其中 ZR 是在带头位置正右边写值 0 的宏图灵机 (练习 6)。一个有着一个输入参数以及两个局部变量的初始化过程, 将会产生如右表所示的图灵机格局。其中 i 是该计算中输入参数的值。带头所在的位置使用下划线进行标记。

在 TM 中, LOAD 和 STOR 指令用来访问以及存储变量的值。这些指令的目标是使得内存管理对于用户来说是透明的。在图灵机中并没有给出带数量的上边界, 而这部分带则有可能被要求存储变量的值。由于分配给每一个变量的带位置的数量并没有被预先分配好, 因此会使得图灵机的内存管理变得十分的复杂。这种忽略其实是有目的的, 即, 为了给予图灵机计算最大的灵活性。那种常用的编译器所给出的标准方法——对于每个变量分配一个固定大小的内存——会导致在需要存储的值超过预分配大小内存时的溢出错误。

指令	格局
	$\underline{B}iB$
INIT2	$\underline{B}iB \bar{O}B$
INIT3	$\underline{B}iB \bar{O}B \bar{O}B$
HOME3	$B\bar{i}B\bar{O}B\bar{O}B$

STOR 指令取出寄存器 t 中的值并存储到特定的变量位置中。这条命令仅用于当 t 是具有分配的值的最大寄存器。在将寄存器 t 的值存储到变量 v_i 中时, 必须维护所有变量的正确的空间。图灵机在实现 STOR 命令的时候主要使用了宏图灵机 INT, 从而能够将值从寄存器中移动到正确的位置。宏图灵机 INT 被假定为停留在带片断 $\underline{B}x\bar{B} \bar{y}B$ 中 (练习 6)。

命令 STOR 如下定义:

指令	定义
STOR $v_i, 1$	$\left(\begin{array}{c} ML_1 \\ INT \end{array} \right)^{n-i+1}$ $\left(\begin{array}{c} MR_1 \\ INT \end{array} \right)^n$ MR_1 ER_1

[316]

指令	定义
STOR v_i, t	MR_{t-2} INT $\left(\begin{array}{c} ML_1 \\ INT \end{array} \right)^{t+n-i-1}$ $\left(\begin{array}{c} MR_1 \\ INT \end{array} \right)^{t+n-i-1}$ MR_1 ER_1 ML_{t-1}

其中 $t > 1$ 且 n 是输入参数以及局部变量的总数, 指数 $n-i+1$ 和 $n-i$ 指出了宏图灵机的循环次数。当寄存器 t 的值被保存后, 寄存器被擦除。

在图灵机执行指令 $\text{STOR } v_2$ 所得到的格局中, 追踪 1 从而来显示在 TM 的内存管理中宏图灵机的角色。在执行指令之前, 带头在中心位置。

图灵机对 LOAD 指令的实现仅仅是通过拷贝变量 v_i 的值到特定的寄存器中来实现的。

指令	定义
$\text{LOAD } v_i, t$	ML_{n-i+1}
	$\text{CPY}_{1, n-i+1+t}$
	MR_{n-i+1}

前面提到, 将一个值加载到寄存器 t 中, 需要寄存器 $1, 2, \dots, t-1$ 都已经被填满了。因而, 为了指令 $\text{LOAD } v_i$ 的正确执行, 图灵机一定处于下面的状态

$$B\bar{v}_1 \ B\bar{v}_2 \ B \dots B\bar{v}_k \ B\bar{v}_{k+1} \ B \dots B\bar{v}_n \ B\bar{r}_1 \ B\bar{r}_2 \ B \dots B\bar{r}_{t-1} \ B$$

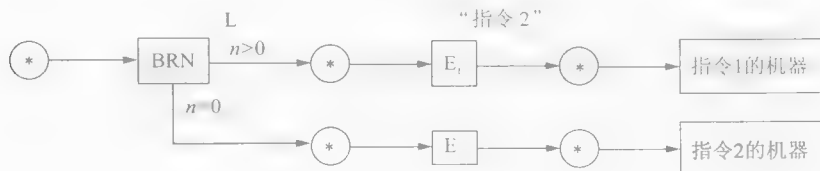
指令 RETURN 与 CLEAR 重新配置了带从而获得计算的结果。如果当指令 $\text{RETURN } v_i$ 运行时, 带头处于中心位置, 并且没有任何寄存器被占用的话, 则带被重写, 并且将 v_i 的值写到图灵机输出的位置上。 CLEAR 指令则是仅将寄存器中的值删除。

代数操作修改寄存器中的值。 INC 、 DEC 以及 ZERO 依次用后继图灵机、前驱图灵机以及零图灵机来定义。为了汇编语言 TM 所定义的其他代数操作, 可以通过创建新的图灵机来实现。比如说, 汇编语言中指令 ADD 能够使用例 9.2.1 中可以实现加法的图灵机 A 来定义, 那么指令 ADD 则能够将寄存器 1 和寄存器 2 中的值相加, 并且将结果存储在寄存器 1 中。既然我们能够通过增加更多的代数操作来大量增加汇编语言的指令数, 那么 INC 、 DEC 以及 ZERO 对于开发语言的目的来说就是足够的了。

汇编语言指令的执行包括定义每条指令的图灵机和宏图灵机的顺序操作。 BRN 以及 GOTO 指令则打破了这种顺序执行, 因为它们可以明确地指定下一条要执行的指令。 GOTO L 指出了标记为 L 的指令将是要执行的下一条指令。分支指令 $\text{BRN L}, t$, 在指出下一条顺序执行的指令之前会测试寄存器 t , 如果寄存器中的值不是 0, 则这个分支指令下面紧挨着的指令就是下一条要被执行的, 否则, 标志为 L 的指令为下一条要被执行的。图灵机中对分支的实现由下图描述:

$\text{BRNL}, 1$

“指令 1”



图灵机	格局
	$B \ \bar{v}_1 \ B \ \bar{v}_2 \ B \ \bar{v}_3 \ B \ \bar{r} \ B$
ML_1	$B \ \bar{v}_1 \ B \ \bar{v}_2 \ B \ \bar{v}_3 \ B \ \bar{r} \ B$
INT	$B \ \bar{v}_1 \ B \ \bar{v}_2 \ B \ \bar{r} \ B \ \bar{v}_3 \ B$
ML_1	$B \ \bar{v}_1 \ B \ \bar{v}_2 \ B \ \bar{r} \ B \ \bar{v}_3 \ B$
INT	$B \ \bar{v}_1 \ B \ \bar{r} \ B \ \bar{v}_2 \ B \ \bar{v}_3 \ B$
MR_1	$B \ \bar{v}_1 \ B \ \bar{r} \ B \ \bar{v}_2 \ B \ \bar{v}_3 \ B$
INT	$B \ \bar{v}_1 \ B \ \bar{r} \ B \ \bar{v}_3 \ B \ \bar{v}_2 \ B$
MR_1	$B \ \bar{v}_1 \ B \ \bar{r} \ B \ \bar{v}_3 \ B \ \bar{v}_2 \ B$
E_1	$B \ \bar{v}_1 \ B \ \bar{r} \ B \ \bar{v}_3 \ B \ B$

指令	定义
$\text{RETURN } v_i$	ML_n
	E_{i-1}
	T
	MR_1
	FR
CLEAR_t	E_{n-i+1}
	FL
	MR_{t-1}
	E_1
	ML_{t-1}

[317]

[318]

测试完该值，寄存器被清空，然后定义了适当操作的图灵机即开始执行。

例 9.6.1 有着一个输入变量和两个局部变量的 TM 程序定义如下，它计算函数 $f(n) = 2n + 1$ ，输入变量是 v_1 且计算中需要用到的局部变量是 v_2 和 v_3 。

```

INIT  $v_2$ 
INIT  $v_3$ 
HOME 3
LOAD  $v_1, 1$ 
STOR  $v_2, 1$ 
L1    LOAD  $v_2, 1$ 
      BRN L2, 1
      LOAD  $v_1, 1$ 
      INC
      STOR  $v_1, 1$ 
      LOAD  $v_2, 1$ 
      DEC
      STOR  $v_2, 1$ 
      GOTO L1
L2    LOAD  $v_1, 1$ 
      INC
      STOR  $v_1, 1$ 
      RETURN  $v_1$ 

```

变量 v_2 被用来作为一个计数器，它在从标志 L1 到 GOTO 语句的每一次循环中递减 1，在每一次循环中， v_1 的值递增，循环会持续 n 次，其中 n 即是输入。循环一结束，值又增加一次，而带上留下的结果则是 $2v_1 + 1$ 。□

构造 TM 汇编语言的目的是用来说明图灵机的指令，和传统的机器一样，是能够被形式化成为高级语言中的命令。利用编程语言中的定义和编译方法，高级语言的命令能够由一系列汇编语言的指令来定义。这将会让图灵机的计算与我们熟悉的算法系统更加相似。

319

9.7 练习

- 若输入字母表为 $\{a, b\}$ ，请构造图灵机来计算一些特定的函数。符号 u 和 v 代表了任意的 $\{a, b\}^*$ 的字符串。
 - $f(u) = aaa$
 - $f(u) = \begin{cases} a & \text{如果 } u \text{ 的长度是偶数} \\ b & \text{其他情况} \end{cases}$
 - $f(u) = u^R$
 - $f(u, v) = \begin{cases} u & \text{如果 } u \text{ 比 } v \text{ 长} \\ v & \text{其他情况} \end{cases}$
- 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ 是一个能够计算语言 L 的部分特征函数的图灵机。那么，请使用图灵机 M 来构造一个能够接收语言 L 的图灵机。
- 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个接收语言 L 的标准图灵机。构造一个能够计算 L 的部分特征函数的图灵机 M' 。一定要注意，当 M' 计算 x_1 完成后，图灵机 M' 带上面的数据应该有这样的形式： $q_f B 0 B$ 或者 $q_f B 1 B$ 。
- 设 L 是字母表 Σ 上的语言，设

$$\chi_L(w) = \begin{cases} 1 & \text{若 } w \in L \\ 0 & \text{其他情况} \end{cases}$$

是语言 L 的特征函数。

- a) 如果 χ_L 是图灵可计算的, 证明 L 是递归的。
 - b) 如果 L 是递归的, 证明存在一个图灵机能够计算 χ_L 。
5. 构造能够计算下列数论函数以及关系的图灵机。不要使用宏图灵机。
- a) $f(n) = 2n + 3$
 - b) $half(n) = \lfloor n/2 \rfloor$ 其中 $\lfloor x \rfloor$ 是比 x 小或者相等的最大整数
 - c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
 - d) $even(n) = \begin{cases} 1 & \text{如果 } n \text{ 是偶数} \\ 0 & \text{其他情况} \end{cases}$
 - e) $eq(n, m) = \begin{cases} 1 & \text{如果 } n = m \\ 0 & \text{其他情况} \end{cases}$
 - f) $lt(n, m) = \begin{cases} 1 & \text{如果 } n < m \\ 0 & \text{其他情况} \end{cases}$
 - g) $n - m = \begin{cases} n - m & \text{如果 } n \geq m \\ 0 & \text{其他情况} \end{cases}$
6. 构造一个图灵机, 它能够执行由下列宏定义的操作, 而且计算的过程中不能离开输入格局中的带片断。
- a) ZR; 输入为 $\underline{B}BB$, 输出为 $\underline{B}\bar{0}B$
 - b) FL; 输入为 $B \bar{n}B' \underline{B}$, 输出为 $\underline{B}\bar{n}B' B$
 - c) E_2 ; 输入为 $\underline{B}\bar{n}B \bar{m}B$, 输出为 $\underline{B}\bar{B}^{n+m+3}B$
 - d) T; 输入为 $\underline{B}B' \bar{n}B$, 输出为 $\underline{B}\bar{n}B' B$
 - e) BRN; 输入为 $\underline{B}\bar{n}B$, 输出为 $\underline{B}\bar{n}B$
 - f) INT; 输入为 $\underline{B}\bar{n}B \bar{m}B$, 输出为 $\underline{B}\bar{m}B \bar{n}B$
7. 使用 9.2 节到 9.4 节中的宏图灵机以及已经构造的图灵机来设计能够计算下面函数的图灵机:
- a) $f(n) = 2n + 3$
 - b) $f(n) = n^2 + 2n + 2$
 - c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
 - d) $f(n, m) = m^3$
 - e) $f(n_1, n_2, n_3) = n_2 + 2n_3$
8. 设计能够计算下列关系的图灵机。你可以使用 9.2 节到 9.4 节中的宏图灵机以及已经构造的图灵机, 还能够使用练习 5 中的图灵机:
- a) $gt(n, m) = \begin{cases} 1 & \text{若 } n > m \\ 0 & \text{其他} \end{cases}$
 - b) $persq(n) = \begin{cases} 1 & \text{若 } n \text{ 是完全平方数} \\ 0 & \text{其他} \end{cases}$
 - c) $divides(n, m) = \begin{cases} 1 & \text{若 } n > 0, m > 0 \text{ 且 } m \text{ 能够整除 } n \\ 0 & \text{其他} \end{cases}$
9. 如果是如下的输入参数, 请写出图灵机 MULT 的动作序列:
- a) $n = 0, m = 4$
 - b) $n = 1, m = 0$
 - c) $n = 2, m = 2$
10. 描述下面每一个合成函数的映射:

a) $\text{add} \circ (\text{mult} \circ (\text{id}, \text{id}), \text{add} \circ (\text{id}, \text{id}))$

b) $p_1^{(2)} \circ (s \circ p_1^{(2)}, e \circ p_2^{(2)})$

c) $\text{mult} \circ (c_2^{(3)}, \text{add} \circ (p_1^{(3)}, s \circ p_2^{(3)}))$

d) $\text{mult} \circ (\text{mult} \circ (p_1^{(1)}, p_1^{(1)}), p_1^{(1)})$

11. 给出满足下面的条件的一元数论全函数:

a) g 不是 id , h 也不是 id , 但是 $g \circ h = \text{id}$ 。

b) g 不是一个常数函数, h 也不是一个常数函数, 但是 $g \circ h$ 却是一个常数函数

12. 给出满足下面条件的一元数论函数:

a) g 不是一对一的函数, h 不是一个全函数, 但是 $h \circ g$ 是一个全函数;

b) $g \neq e$, $h \neq e$, $h \circ g = e$, 其中 e 是空函数。

c) $g \neq \text{id}$, $h \neq \text{id}$, $h \circ g = \text{id}$, 其中 id 是恒等函数。

d) g 不是全函数, h 不是一对一的函数, 但是 $h \circ g = \text{id}$ 。

* 13. 设 F 是一个能够计算一元数论函数 f 的图灵机。设计一个能够返回第一个自然数 n 的图灵机, 例如 $f(n) = 0$ 。如果不存在这样的 n 则计算将会不确定地继续。如果 F 所计算的函数不是全函数, 那么会发生什么样的情况?

14. 设 F 是一个能够计算一元数论函数 f 的图灵机。设计一个能够计算如下函数

$$g(n) = \sum_{i=0}^n f(i)$$

的图灵机。

15. 设 F 和 G 是分别能够计算一元数论函数 f 和 g 的图灵机。设计一个能够计算下面的函数

$$h(n) = \sum_{i=0}^n eq(f(i), g(i))$$

的图灵机。也就是说, $h(n)$ 是在范围 0 到 n 之间的函数 f 和 g 的具有同样值的次数

16. 在 N 上的一元关系 R 是图灵可计算的, 如果其特征函数是可计算的。证明 N 上的任意可计算一元关系都能够定义一个递归语言。提示: 由计算其特征函数的图灵机构造出接收 R 的图灵机

* 17. 设 $R \subseteq \{1\}^*$ 是一个递归语言, 证明 R 定义了一个 N 上的可计算的一元关系。

18. 证明在 N 上的所有一元关系中, 存在不是图灵可计算的一元关系。

322 19. 设 F 是一个包含了所有一元数论全函数的集合, 这些函数满足 $f(i) = i$, 且 i 是任意一个为偶数的自然数。证明 F 中存在不是图灵可计算的函数。

20. 设 v_1, v_2, v_3, v_4 是一个 TM 程序中所需要用到变量, 设寄存器 1 保存了一个值, 请写出在执行指令 $\text{STOR } v_2, 1$ 时的动作。可以参看例 9.3.2 中介绍的方法来写出这些动作。

21. 写出一个计算函数 $f(v_1, v_2) = v_1 \div v_2$ 的 TM 程序。

参考文献注释

图灵机汇编语言提供了一个体系结构, 这个体系结构与另外一系列的抽象机器——随机访问机器——很相似 [Cook 和 Reckhow, 1973]。随机访问机器包含了无穷个内存单元位置以及有限个数的寄存器, 每一个寄存器都能够存储一个整数。而随机访问机器的指令则能够操作寄存器和内存并且还能够执行代数操作。这些机器是标准冯诺伊曼计算机体系结构的抽象。在 Aho、Hopcroft 和 Ullman [1974] 中能够找到随机访问机器及其等价的图灵机的详细介绍。

第 10 章 乔姆斯基层次

在第 3 章中,我们将正则文法以及上下文无关文法介绍成基于规则的系统,主要用来生成某种语言的字符串。每条规则定义了一个字符串的转换,任意语言的一条语句都是使用一系列允许的字符串转换得到的。正则文法以及上下文无关文法实际上是某个更为泛化的短语结构文法的子集。短语结构文法被诺姆·乔姆斯基作为自然语言的语法模型而提出。在本章中,我们将会探讨另外的两类短语结构文法——无限制文法和上下文有关文法。这四种文法——正则文法、上下文无关文法、上下文有关文法以及无限制文法,组成了乔姆斯基关于短语结构文法的层次,而层次中的每一种后继的类则会在定义一个规则的时候允许额外的灵活性。

自动机被设计成为机械地识别正则语言以及上下文无关语言;确定型有限自动机则接收由正则文法产生的语言,而下推自动机接收上下文无关文法产生的语言。文法的产生与机械的接收之间的关系被扩展成了新类型的文法。图灵机就被证明能够接收由无限制文法所产生的语言。通过限制图灵机的内存大小所获得的一类机器则能够接收由上下文有关文法所产生的语言。

10.1 无限制文法

短语结构文法被设计成提供自然语言语法的形式化模型。其名字(短语结构)基于如下命题:属于某一语言的句子有可能存在几种不同的语法模式。句子本身是由短语组成的;名词短语、动词短语以及类似的短语,它们都是按照某一个句子模式来专门安排的。语法的规则定义了句子以及短语的结构。

[325]

短语结构文法的成分和第 3 章中介绍的正则文法和上下文无关文法的成分是一样的。短语结构文法包含了一个变量的有限集合 V 、一个字母表 Σ 、一个初始变量以及一套规则集。这些规则的形式均为 $u \rightarrow v$, 其中 u 和 v 包含了变量以及终止符号,并且定义了所能允许的字符串转换。对字符串 z 应用某一条规则是两步的过程,包括:

- i) z 的某一个子串符合规则的左半部分,然后
- ii) 用规则的右半部分替换左半部分。

对字符串 xuy 应用规则 $u \rightarrow v$, 写为 $xuy \Rightarrow xvy$, 会产生了字符串 xvy 。如果存在一系列的规则能够将 p 转换成 q , 则说明字符串 q 能够由 p 推导出, 即 $p \Rightarrow^* q$ 。 G 的语言, 记作 $L(G)$, 是能够被开始符号 S 推导出的, 使用终止符号组成的字符串的集合。用符号来表示就是 $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ 。

一个类型的语法都是由对规则集的形式进行某种约束来定义的。而上下文无关文法也是短语结构文法, 只不过上下文无关文法的每一条规则的左半部是一个单独的变量, 而右半边则能够是变量和终结符号的任意组合。正则文法的每一条规则都必须有如下的形式:

- i) $A \rightarrow aB$,
- ii) $A \rightarrow a$, 或者
- iii) $A \rightarrow \lambda$,

其中 $A, B \in V$, 而 $a \in \Sigma$ 。

无限制文法是短语结构文法中最大的一类, 对于无限制文法的规则, 没有任何的限制, 只是要求规则的左边不能为空而已。

定义 10.1.1 无限制文法是一个四元组 (V, Σ, P, S) , 其中 V 是变量的一个有限集合; Σ (字母表) 是终止符号的有限集合; P 是规则的集合; 而 S 是 V 集合中的一个特殊的元素。无限制文法的产生式有下面的形式: $u \rightarrow v$, 其中 $u \in (V \cup \Sigma)^+$, $v \in (V \cup \Sigma)^*$, 且集合 V 和 Σ 不相交。

下面有两个例子来描述无限制文法的生成能力。例 10.1.1 描述了语言 $\{a^i b^i c^i \mid i \geq 0\}$, 我们知道该语言是不可以从上下文无关文法推导来的, 但是却能够由无限制文法通过六条规则来生成。第二个

[326]

例子描述了无限制文法是怎样用来生成字符串的拷贝的。

例 10.1.1 无限制文法

$$\begin{aligned} V &= \{S, A, C\} & S &\rightarrow aAbc \mid \lambda \\ \Sigma &= \{a, b, c\} & A &\rightarrow aAbC \mid \lambda \\ & & Cb &\rightarrow bC \\ & & Cc &\rightarrow cc \end{aligned}$$

能够生成语言 $\{a^i b^j c^i \mid i \geq 0\}$ ，其中开始符号是 S 。字符串 $a^i b^j c^i$ ，其中 $i > 0$ ，是能够由开始符号 S 推导的，具体过程如下：

$$\begin{aligned} S &\Rightarrow aAbc \\ &\Rightarrow a^i A (bC)^{i-1} bc \\ &\Rightarrow a^i (bC)^{i-1} bc \end{aligned}$$

使用规则 $A \rightarrow aAbC$ 能够生成最开始的 i 个 a ，而规则 $Cb \rightarrow bC$ 则允许最后的一个 C 能够与 b 进行位置的交换，从而越过一个个的 b 从而到达 b 和 c 的分界处，当到达最左边的 c 的时候，符号 C 能够用 c 来替换。这个过程一直持续到每一个 C 都往右移动越过所有的 b ，并且最终被转换成 c 。□

例 10.1.2 无限制文法，其终止符号字母表为 $\{a, b, [,]\}$ ，由下列产生式组成：

$$\begin{aligned} S &\rightarrow aT[a] \mid bT[b] \mid [] \\ T[&\rightarrow aT[A] \mid bT[B] \mid [\\ Aa &\rightarrow aA \\ Ab &\rightarrow bA \\ Ba &\rightarrow aB \\ Bb &\rightarrow bB \\ A] &\rightarrow a] \\ B] &\rightarrow b] \end{aligned}$$

产生语言 $\{u[u] \mid u \in \{a, b\}^*\}$ 。

变量 T 左边多出的 a 或者 b 则会在使用产生式后，在 $T[$ 右边生成变量 A 或者 B 。使用这些规则交换了一个变量和一个终结符的位置，而推导的过程则是通过传递在括号中的字符串的拷贝进行的。当变量与符号 $]$ 相邻的时候，恰当的终结符就被加到第二个字符串中去了。整个过程重复着来生成其他的终结符，或者通过应用规则 $T[\rightarrow [$ 来终止。而推导过程

$$\begin{aligned} S &\Rightarrow aT[a] \\ &\Rightarrow aaT[Aa] \\ &\Rightarrow aaT[aA] \\ &\Rightarrow aaT[aa] \\ &\Rightarrow aabT[Baa] \\ &\Rightarrow aabT[aBa] \\ &\Rightarrow aabT[aaB] \\ &\Rightarrow aabT[aab] \\ &\Rightarrow aab[aab] \end{aligned}$$

则展示了在推导过程中变量所扮演的角色。□

在上述两个例子的文法中，每一条规则的左半边包含了一个变量，这并不是无限制文法所要求的。然而，如果假设必须要在规则的左半边包含一个变量的话，也并不会缩小要产生的语言的集合（练习 3）。

在对整个形式语言的研究中，对于一个语言来说，我们描述了通过文法产生以及使用一个有限状态机来接收它们之间的对应。正规语言使用一个有限状态自动机来接收；而上下文无关语言则使用下推自动机来接收。无限制文法提供了字符串转换的最灵活的典型，对于需要匹配的子串及其的替换字

字符串,没有做出任何的限制。那么,无限制文法所能产生的语言需要一个最有能力的抽象机器来接收,这一点看上去是十分合理的,而事实也恰是如此。下面的两个定理说明了一个语言可由无限制文法产生当且仅当其能够被一个图灵机所接收。

定理 10.1.2 设 $G = (V, \Sigma, P, S)$ 是一个无限制文法, 则语言 $L(G)$ 是一个递归可枚举语言。

证明: 我们简要描述一下设计一个三带非确定型图灵机 M 来接收语言 $L(G)$ 。我们设计图灵机 M , 它计算模拟了文法 G 的推导过程。第一带上面保存了输入字符串 p , 其中 $p \in \Sigma^*$, 第二带上面则保存了 G 中的规则的表示形式, 其中每一条规则 $u \rightarrow v$ 使用 $u\#v$ 来表示, 其中 $\#$ 是为了分隔 u, v 而专门保存的带符号。规则之间用两个连续的 $\#$ 符号来分隔。而对于文法 G 的模拟则在第三带上进行。

接收语言 $L(G)$ 的图灵机 M 的计算包括下面的动作:

1. 开始符号 S 被写在第三带第一个位置上;
2. 文法 G 的规则写在第二带上;
3. 从第二代上面选取一个规则 $u\#v$;
4. 如果存在的话, 就在第三带上面选择字符串 u 的一个实例, 否则, 即不存在这样的 u , 则计算终止, 且处于拒绝状态;
5. 在第三带上用 v 来替代 u ;
6. 如果第三带上面的字符串和第一带上的匹配, 那么计算终止, 且处于接收状态;
7. 计算从第三步开始继续, 模拟其他规则的应用。

由于 u 和 v 的长度有可能不同, 则应用规则 $xyu \rightarrow xvu$ 则可能需要将字符串 v 向左或者向右平移。

对于任意的字符串 $p \in L(G)$, 需要应用一系列的规则来推导出 p 。这种推导需要图灵机 M 的某次非确定型的计算来检查, 且 M 将会接收 p 。相反地, 图灵机在第三带上的动作从开始符号 S 开始, 精确地产生了字符串。图灵机 M 所能够接收的是语言 $L(G)$ 中的终结符号组成的字符串。因而, $L(M) = L(G)$ 。

例 10.1.3 语言 $L = \{a^i b^j c^i \mid i \geq 0\}$ 由下面规则产生

$$\begin{aligned} S &\rightarrow aAbc \mid \lambda \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

接收语言 L 的图灵机的计算模拟了文法的推导。在第二带上的文法规则的表示是:

$$BS\#aAbc\#\#\#\#\#A\#aAbC\#\#\#\#\#Cb\#bC\#\#\#Cc\#ccB$$

规则 $S \rightarrow \lambda$ 用字符串 $S\#\#\#$ 来表示。第一个 $\#$ 符号将推导规则的左半边和右半边分隔开来, 规则的右半边, 这个例子中是空字符串, 后面紧跟着字符串 $\#\#\#$ 。

定理 10.1.3 设 L 是一个递归可枚举语言, 则存在一个无限制文法 G , 且 $L(G) = L$ 。

证明: 由于 L 是一个递归可枚举语言, 则它能够使用一个确定型图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 来接收。那么使用一个无限制文法 $G = (V, \Sigma, P, S)$ 可以模拟图灵机 M 的运算。将图灵机的配置参数用字符串来表示, 则图灵机的一个状态转移 $\delta(q_i, x) = [q_j, y, R]$, 配置参数为 $uq_i xvB$, 能够使用字符串形式 $uq_i xvB \Rightarrow uyq_j vB$ 来表示。

在文法 G 中, 一个终结符字符串的推导包括了三个不同的子推导:

- i) 生成字符串 $u[q_0 Bu]$, 其中 $u \in \Sigma^*$;
- ii) 模拟图灵机 M 在字符串 $[q_0 Bu]$ 之上的运算, 且
- iii) 如果图灵机 M 接收 u , 则移除模拟的子字符串。

对于任意一个终结符 $a_i \in \Sigma$, 文法 G 均包含一个变量 A_i 。这些变量与 $S, T, [,]$ 和 ϵ 一起用来生成字符串 $u[q_0 Bu]$ 。对于某个运算的模拟则使用与 M 当前状态相对应的变量。变量 E_R 和 E_L 用在推导的第三步中。文法的终结符是图灵机 M 的输入字母表的一个元素, 因而文法 G 的字母表是:

$$\begin{aligned} \Sigma &= \{a_1, a_2, \dots, a_n\} \\ V &= \{S, T, E_R, E_L, [,], A_1, A_2, \dots, A_n\} \cup Q \end{aligned}$$

推导中的每一部分的规则单独给出。推导往往开始于生成 $u[q_0Bu]$, 其中 u 是 Σ^* 中的任意一个字符串, 生成这种形式的字符串的策略在例 10.1.2 中有所介绍。

$$1. S \rightarrow a_i T[a_i] [q_0 B] \quad 1 \leq i \leq n$$

$$2. T[\rightarrow a_i T[A_i] [q_0 B] \quad 1 \leq i \leq n$$

$$3. A_i a_j \rightarrow a_j A_i \quad 1 \leq i, j \leq n$$

$$4. A_i] \rightarrow a_i] \quad 1 \leq i \leq n$$

当输入字符串是 u 时, 图灵机的计算是在字符串 $[q_0Bu]$ 上模拟的, 并且通过重写图灵机 M 的状态转换来得到新规则, 这些规则主要是用来说明字符串的转换。

$$5. q_i xy \rightarrow zq_j y \quad \text{当 } \delta(q_i, x) = [q_j, z, R] \text{ 且 } y \in \Gamma$$

$$6. q_i x] \rightarrow zq_j B] \quad \text{当 } \delta(q_i, x) = [q_j, z, R]$$

$$7. yq_i x \rightarrow q_j yz \quad \text{当 } \delta(q_i, x) = [q_j, z, L] \text{ 且 } y \in \Gamma$$

如果图灵机 M 的运算在接收状态停机, 则推导中擦除了括号内的字符串。变量 E_R 擦除了停机位置带头右边的字符串。当到达了终端的标记符号 $\}$ 时, 就产生了 E_L 这个变量。

$$8. q_i x \rightarrow E_R \quad \text{当 } \delta(q_i, x) \text{ 未被定义且 } q_i \in F$$

$$9. E_R x \rightarrow E_R \quad x \in \Gamma$$

$$10. E_R] \rightarrow E_L$$

$$11. xE_L \rightarrow E_L \quad x \in \Gamma$$

$$12. [E_L \rightarrow \lambda$$

推导从生成字符串 $u[q_0Bu]$ 开始, 终止于产生了字符串 u , 且 $u \in L(M)$ 。如果 $u \notin L(M)$, 则嵌套在推导过程中的括号永远也不会被删除, 且推导的过程不能够产生一个终结符字符串。

例 10.1.4 构造一个能产生被下图所示的图灵机接收的语言的文法, 该图灵机接收语言 $a^*b(a \cup b)^*$ 。当遇到第一个 b 时, 图灵机 M 停机, 且此时的状态是 q_1 。

文法 G 的变量和终结符是

$$\Sigma = \{a, b\}$$

$$V = \{S, T, E_R, E_L, [,], A, X\} \cup \{q_0, q_1\}。$$

规则分为三部分给出。

331 模拟规则:

转换	规则
$\delta(q_0, B) = [q_1, B, R]$	$q_0 Ba \rightarrow Bq_1 a$ $q_0 Bb \rightarrow Bq_1 b$ $q_0 BB \rightarrow Bq_1 B$ $q_0 B] \rightarrow Bq_1 B]$
$\delta(q_1, a) = [q_1, a, R]$	$q_1 aa \rightarrow aq_1 a$ $q_1 ab \rightarrow aq_1 b$ $q_1 aB \rightarrow aq_1 B$ $q_1 a] \rightarrow aq_1 B]$
$\delta(q_1, B) = [q_1, B, R]$	$q_1 Ba \rightarrow Bq_1 a$ $q_1 Bb \rightarrow Bq_1 b$ $q_1 BB \rightarrow Bq_1 B$ $q_1 B] \rightarrow Bq_1 B]$

输入生成规则:

$$S \rightarrow aT[a] \mid bT[b] \mid [q_0 B]$$

$$T[\rightarrow aT[A] \mid bT[X] [q_0 B]$$

$$Aa \rightarrow aA$$

$$Ab \rightarrow bA$$

$$A] \rightarrow a]$$

$$Xa \rightarrow aX$$

$$Xb \rightarrow bX$$

$$X] \rightarrow b]$$

删除规则:

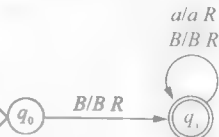
$$q_1 b \rightarrow E_R$$

$$E_R a \rightarrow E_R$$

$$E_R b \rightarrow E_R$$

$$E_R B \rightarrow E_R$$

$$E_R] \rightarrow E_L$$



$$aE_L \rightarrow E_L$$

$$bE_L \rightarrow E_L$$

$$BE_L \rightarrow E_L$$

$$[E_L \rightarrow \lambda$$

在图灵机 M 中, 输入字符串为 ab , 则被 M 接收, 在文法 G 中有相同的推导用于接收文法 G , 其推导

过程如下:

$$\begin{array}{ll}
 q_0 BabB & S \Rightarrow aT[a] \\
 \vdash Bq_1 abB & \Rightarrow abT[Xa] \\
 \vdash Baq_1 bB & \Rightarrow ab[q_0 BXa] \\
 & \Rightarrow ab[q_0 BaX] \\
 & \Rightarrow ab[q_0 Bab] \\
 & \Rightarrow ab[Bq_1 ab] \\
 & \Rightarrow ab[Baq_1 b] \\
 & \Rightarrow ab[BaE_R] \\
 & \Rightarrow ab[BaE_L] \\
 & \Rightarrow ab[BE_L] \\
 & \Rightarrow ab[E_L] \\
 & \Rightarrow ab.
 \end{array}$$

无限制文法的属性能够用于建立递归可枚举语言的封闭结果。这个证明与定理 7.5.1 中对上下文无关文法的证明类似, 并留作练习题。□

定理 10.1.4 递归可枚举语言的集合在并运算、连接运算以及克林星闭包运算上是封闭的。

10.2 上下文有关文法

上下文有关文法代表了上下文无关文法和无限制文法的中间状态。它对于产生式的左半部没有任何限制, 但是右半部的长度要求最起码和左半部一样长 (即要长于左半部分或者和左半部分一样长)。

332

定义 10.2.1 短语结构文法 $G = (V, \Sigma, P, S)$ 被称为上下文有关文法, (context-sensitive) 当文法 G 中的每一条形如 $u \rightarrow v$ 的规则, 其中 $u \in (V \cup \Sigma)^*$, $v \in (V \cup \Sigma)^*$, 都有 $\text{length}(u) \leq \text{length}(v)$ 。

满足定义 10.2.1 中的规则被称为单调的 (monotonic)。每一次对单调规则的应用, 都会使得推导后的字符串的长度或者相等, 或者增加。上下文有关文法产生的语言, 被称为上下文有关语言。

上下文有关文法最初如同短语结构文法一样来定义, 该文法中的每一条规则都形如 $uAv \rightarrow uwv$, $A \in V, w \in (V \cup \Sigma)^*$ 且 $u, v \in (V \cup \Sigma)^*$ 。上面的规则表明了变量 A 能够用 w 来替换, 当且仅当其处于前面有一个 u , 后面有一个 v 的上下文之中。很明显, 每一条这样定义的规则都是单调的。从另一方面来说, 由一条单调规则定义的状态转换能够使用一组形如 $uAv \rightarrow uwv$ 的规则来生成 (练习 10 和练习 11)。

规则的单调属性保证了空字符串不是上下文有关语言的元素。将例 10.1.1 中的规则 $S \rightarrow \lambda$ 去掉, 则得到了无限制文法

$$\begin{array}{l}
 S \rightarrow aAbc \\
 A \rightarrow aAbC \mid \lambda \\
 Cb \rightarrow bC \\
 Cc \rightarrow cc
 \end{array}$$

该文法能够产生语言 $\{a^i b^i c^i \mid i > 0\}$ 。由于 λ 规则违反了上下文有关文法的单调性, 所以则将 S 规则和 A 规则用下面的规则来替换

$$\begin{array}{l}
 S \rightarrow aAbc \mid abc \\
 A \rightarrow aAbC \mid abC
 \end{array}$$

这样就产生了一个等价的上下文有关文法。

和定理 10.1.2 中图灵机类似, 设计一个非确定型图灵机来接收上下文有关语言。规则中确立的输入字符串只能增长的特性, 可以允许使用输入字符串的长度来终止不成功推导的模拟过程。当推导出的字符串的长度比输入字符串的长度长的时候, 计算终止, 并且拒绝该输入字符串。

定理 10.2.2 每一个上下文有关语言都是递归的。

[333] 证明: 使用定理 10.1.2 中提出的方法, 在一个三带非确定型图灵机 M 上模拟上下文有关文法的推导。整个推导过程, 包括最后的结果, 都保存在第三带上。当规则 $u \rightarrow v$ 应用在第二带上的字符串 xuy 的时候, 字符串 xvy 被写在带上, 并紧跟在字符串 $xuy\#$ 之后。符号 $\#$ 被用来分隔推导的字符串。

当输入字符串为 p 时, 图灵机 M 的计算将会执行下述的一系列的操作:

1. $S\#$ 被写在第三带的第一个位置上。
2. 文法 G 的规则写在第二带上。
3. 从第二带上选择规则 $u \rightarrow v$ 。
4. 设 $q\#$ 是第三带上最近写的字符串, 则:
 - a) 如果存在的话, 则在 q 中选择字符串 u 的一个实例。在这个情况之下, q 能被写成 zuy 。
 - b) 否则, 计算则会终止在一个不接收的状态。
5. $xvy\#$ 被写在第三带上, 并紧跟在 $q\#$ 之后。
6. a) 若 $xvy = p$, 则计算终止在接收状态;
 - b) 如果 xvy 出现在第三带的其他位置上, 则计算终止在非接收状态。
 - c) 如果 $\text{length}(xvy) > \text{length}(p)$, 则计算停止在非接收状态。
7. 计算从第三步开始继续, 从而模拟下一条规则的应用。

在 $(V \cup \Sigma)^*$ 中只存在有限个字符串的长度比 $\text{length}(p)$ 短或相等。这表明了每一个推导过程最终都会终止, 比如说进入一个循环, 或者推导一个字符串且该字符串的长度大于 $\text{length}(p)$ 。当选择的规则不能够应用在当前的字符串上时, 有的计算会在第四步终止。循环推导, 如 $S \Rightarrow w \Rightarrow w$, 则会在第 6(b) 步终止。长度的限定用来在第 6(c) 步终止所有不成功的推导。

语言 $L(G)$ 中的每一个字符串都由一个非循环的推导来生成。这样推导的模拟则会导致图灵机 M 接收该字符串。由于图灵机 M 的每一次运算都会终止, 因此说明了 $L(G)$ 是递归的 (练习 8.23)。■

10.3 线性有界自动机

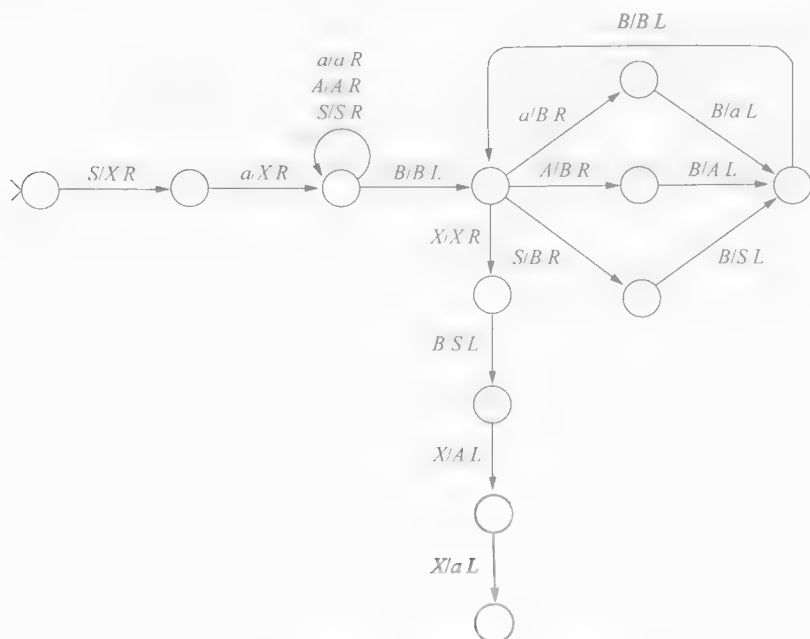
我们已经考察了几种对标准图灵机的修改并不会改变图灵机所接收的语言的集合。对带容量的限制降低了图灵机计算的能力。线性有界自动机是一个图灵机, 但是其带的容量则决定于输入字符串的长度。输入字母表包含两个符号, $<$ 和 $>$, 来表明带的左边界和右边界。

[334] **定义 10.3.1** 线性有界自动机 (linear-bounded automaton, LBA) 是一个 $M = (Q, \Sigma, \Gamma, \delta, q_0, \langle, \rangle, F)$, 其中 $Q, \Sigma, \Gamma, \delta, q_0$ 和 F 与非确定型图灵机中定义的一样。符号 \langle 和 \rangle 是 Σ 中特殊的元素。

LBA 的计算的初始格局是 q_0, w , 需要 $\text{length}(w) + 2$ 个带位置。虽然终端标记符号 \langle 和 \rangle 写在带上, 但并不作为输入字符串的一部分。计算将会由 \langle 和 \rangle 所标记的边界之内进行。终端标记符号能够被机器读取, 但是并不能够被擦除。在读取 \langle 后的状态转换必须有向右的移动, 而读取 \rangle 后的状态转换则必须有向左的移动。如果输入字符串为 $\langle w \rangle$, 且最终图灵机停止在接收状态时, 字符串 $w \in (\Sigma - \{\langle, \rangle\})^*$ 最终能够被 LBA 接收。

我们下面展示每一个上下文有关语言都能够被一个线性有界自动机接收。构造一个 LBA 来模拟上下文有关文法的推导。图灵机能够构造来模拟无限制文法的推导, 且最开始就是将推导的规则写在其中一条带上。而 LBA 的关于带容量的限制则禁止了这种方法。然而, LBA 的状态和状态转换则用来对这些规则进行编码。

图 10-1 表明了应用规则 $Sa \rightarrow aAS$ 时怎样用状态转换来对其进行模拟。规则的应用声明了一个字符串的转换 $uSav \Rightarrow uaASv$ 。这个图中最开始的两个转换确认了带上从带头所在的位置开始的字符串匹配了 Sa 。在 Sa 被 aAS 替换之前, 且在决定推导出的字符串是否与能够参与计算的带片断相适合时, 字符串 v 是不起任何作用的。如果读到了字符 $>$, 则计算终止, 否则, 字符串 v 则向右平移一个单位并且用 aAS 来替换掉 Sa 。

图 10-1 使用 LBA 来模拟 $Sa \rightarrow aAs$

[335]

定理 10.3.2 设 L 是一个上下文有关语言。则存在一个线性有界自动机 M ，且 $L(M) = L$ 。

证明：由于 L 是一个上下文有关语言，则对于某个上下文文法 $G = (V, \Sigma, P, S)$ ，有 $L = L(G)$ 。构造一个具有两道的 LBA M 来模拟文法 G 的推导。第一道包含了输入字符串，且包括了终端字符标记。第二道则存储了由模拟推导产生的字符串。

文法 G 的每条规则都在一个子机器 M 中被编码。当输入是 $\langle p \rangle$ 时，机器 M 包含下面一系列的动作：

1. S 写在第二道的第一个位置。
2. 带头移动到能够扫描第二道上字符串符号的位置。
3. 规则 $u \rightarrow v$ 被非确定地选取，且运算试图应用该条规则。
4. a) 如果第二道上从带头位置开始的子串和 u 不匹配，则计算停止于拒绝接收状态；
b) 如果带头在扫描 u ，但是将 v 替换 u 之后得到的字符串的长度大于 $\text{length}(p)$ ，则运算终止在拒绝接收状态。
c) 否则， u 在第二道上被 v 替换。
5. 如果第二道包含字符串 p ，则计算停止在接收状态。
6. 计算从第二步开始继续进行，从而模拟应用另外一条规则。

定义机器 M 用来接收语言 L 。语言 L 中的每一个字符串都由文法 G 的推导生成，而且推导的模拟导致 M 接收字符串。因而 $L \subseteq L(M)$ 。相反地，当输入字符串是 $\langle p \rangle$ 时， M 的运算终止于接收状态，且运算的过程中包含了由第二步和第三步生成的很多字符串的转换。这些转换定义了文法 G 中 p 的推导和 $L(M) \subseteq L$ 。

为了获得上下文有关语言（线性有界自动机接收语言集合的子集）的完整特性，我们可以展示，被这样一个自动机接收的任意语言都是由上下文有关文法生成的。文法的规则能够直接由自动机的状态转换来构造。

定理 10.3.3 设语言 L 是线性有界自动机接收的语言，则 $L - \{\lambda\}$ 是上下文有关语言。

证明：设 $M = (Q, \Sigma_M, \Gamma, \delta, q_0, \langle \cdot \rangle, F)$ 是接收语言 L 的线性有界自动机。设计一个上下文有关文法 G 来生成 $L(M)$ 。利用定理 10.1.3 中的方法，接收输入字符串 p 的 M 的运算由文法 G 中对 p 的推

[336]

导来模拟。但是这里不能够使用构造模拟图灵机运算的无限制文法的方法,原因是擦除模拟过程的规则并不能够满足上下文有关文法的单调性的限制。在上下文有关文法的推导中,不能够擦除符号的规定限制了推导出的字符串的长度。整个模拟过程通过使用复合对象作为变量来完成。

文法 G 的终止符号字母表由 M 的输入字母表得到,但删除了终端标识符号。有序对被用作变量。有序对的第一个元素是一个终止符号,第二个元素是一个字符串,包含了带符号以及可能的状态和终端标识符。

$$\Sigma_G = \Sigma_M - \{ \langle \rangle, \cdot \} = \{ a_i, a_2, \dots, a_n \}$$

$$V = \{ S, A, [a_i, x], [a_i, \langle x \rangle], [a_i, x], [a_i, \langle x \rangle], [a_i, q_k x], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, q_k x], [a_i, xq_k], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, \langle xq_k \rangle] \},$$

其中 $a_i \in \Sigma_G, x \in \Gamma$, 且 $q_k \in Q$ 。

S 和 A 规则生成了有序对,且有序对的元素代表了输入字符串和 M 的初始配置参数。

$$1. S \rightarrow [a_i, q_0 \langle a_i \rangle] A$$

$$\rightarrow [a_i, q_0 \langle a_i \rangle]$$

其中每一个 $a_i \in \Sigma_G$

$$2. A \rightarrow [a_i, a_i] A$$

$$\rightarrow [a_i, a_i]$$

其中每一个 $a_i \in \Sigma_G$

应用 S 规则和 A 规则推导生成了如下形式的一系列有序对

$$[a_i, q_0 \langle a_i \rangle] \text{ 或}$$

$$[a_i, q_0 \langle a_i \rangle] [a_i, a_i] \dots [a_i, a_i]$$

通过连接有序对的第一个元素得到的字符串 $a_i a_i \dots a_i$ 代表了 M 的一个运算的输入字符串。而第二个元素产生了 $q_0 \langle a_i a_i \dots a_i \rangle$, 即线性有界自动机的初始格局。

模拟图灵机的计算的规则通过重写 M 的状态转换而得到,这些规则也同样是转换,用于修改有序对的第二个元素。一定要注意,有序对的第二个元素并不产生字符串 q_0 , 如果输入字符串是空字符串,则文法也不会对其进行模拟。定理 10.1.3 中的技术能够加以修改从而产生所需规则来模拟 M 的运算,其详细的部分留下作为练习。

337

一旦成功地进行运算,推导过程则必须生成初始的输入字符串。当一个接收配置参数生成的时候,有序对第二个元素中带有接收状态的变量被转换成包含第一个元素的终止符号。

$$3. [a_i, q_k \langle x \rangle] \rightarrow a_i$$

$$[a_i, q_k \langle x \rangle] \rightarrow a_i$$

当 $\delta(q_k, \langle \rangle) = \emptyset$ 且 $q_k \in F$

$$[a_i, xq_k] \rightarrow a_i$$

$$[a_i, \langle xq_k \rangle] \rightarrow a_i$$

当 $\delta(q_k, \langle \rangle) = \emptyset$ 且 $q_k \in F$

$$[a_i, q_k x] \rightarrow a_i$$

$$[a_i, q_k \langle x \rangle] \rightarrow a_i$$

$$[a_i, \langle q_k x \rangle] \rightarrow a_i$$

$$[a_i, \langle q_k \langle x \rangle \rangle] \rightarrow a_i$$

当 $\delta(q_k, x) = \emptyset$ 且 $q_k \in F$

推导将会在把剩下的变量转换成第一个元素中包含的终结符后完成。

$$4. [a_i, u] a_j \rightarrow a_j a_i$$

$$a_j [a_i, u] \rightarrow a_j a_i$$

对于任意的 $a_j \in \Sigma_G$ 且 $[a_i, u] \in V$ 。

10.4 乔姆斯基层次

乔姆斯基列举出了组成乔姆斯基层次的这四类文法（以及语言）：无限制文法、上下文有关文法、上下文无关文法以及正则文法，它们分别被称为0型文法、1型文法、2型文法和3型文法。随着文法前面数字的增加，规则的限制会越来越多。乔姆斯基层次中文法的嵌套导致了相对应的语言的嵌套。每一个包含空字符串的上下文无关语言可以由这样一个上下文无关的文法生成，该文法中 $S \rightarrow \lambda$ 是仅有的包含 λ 的规则（定理4.2.3）。删除这个仅有的 λ 规则会得到一个上下文有关文法，其产生的语言是 $L - \{\lambda\}$ 。因而，语言 $L - \{\lambda\}$ 是上下文有关的，而此时 L 是上下文无关的。忽略由空字符串而给上下文有关语言带来的复杂性，则每一种 i 型语言都是 $i-1$ 型的。

上面的包含是正确的。集合 $\{a^i b^j \mid i \geq 0\}$ 是上下文无关的但并不是正则的（定理6.5.1），类似地， $\{a^i b^j c^k \mid i > 0\}$ 是上下文有关的但却不是上下文无关的（例7.4.1）。在第11章中，停机问题的语言被证明是递归可枚举的但并不是递归的。将这些结果整合在一起，并且应用定理10.2.2，就会建立起上下文有关语言的集合是递归可枚举语言的子集这样一个正确的包含关系了。

在乔姆斯基层次中，每一类语言都由用某一类语法生成的语言以及相应的接收机器的类型来刻画。生成和识别关系综合的描述参见下表。

文 法	语 言	接收的机器
0型文法，短语结构文法，无限制文法	递归可枚举	图灵机，非确定型图灵机
1型文法，上下文有关文法	上下文有关	右线性有界自动机
2型文法，上下文无关文法	上下文无关	下推自动机
3型文法，正则文法，左线性文法，右线性文法	正则	确定型有限自动机，不确定型有限自动机

10.5 练习

1. 设计无限制文法来生成下面的语言：

- $\{a^i b^j a^i b^j \mid i, j \geq 0\}$
- $\{a^i b^j c^k d^l \mid i \geq 0\}$
- $\{www \mid w \in \{a, b\}^*\}$

2. 证明每一个由下列文法

$$\begin{aligned} S &\rightarrow aAbc \mid \lambda \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

生成的终结符字符串都有 $a^i b^j c^i$ 的形式，其中 $i \geq 0$ 。

3. 证明每一个递归可枚举语言能够由一个文法生成，在这个文法中，每一条规则都形如 $u \rightarrow v$ ，其中 $u \in V^*$ ，且 $v \in (V \cup \Sigma)^*$ 。

4. 证明递归可枚举语言对于下面的运算是封闭的：

- 并
- 交
- 连接
- 克林星闭包
- 同态像

5. 设 M 是图灵机

- 给出描述 $L(M)$ 的一个正则表达式。



- b) 使用定理 10.1.3 中的技术, 给出一个接收 $L(M)$ 的无限制文法的规则。
 c) 当输入字符串是 aba 的时候, 写出 M 的运算过程, 并给出在文法 G 中的推导过程。
 6. 设 G 是上下文有关文法:

$$\begin{aligned} G: S &\rightarrow SBA \mid a \\ BA &\rightarrow AB \\ aA &\rightarrow aaB \\ B &\rightarrow b \end{aligned}$$

- a) 给出 $aabb$ 的推导过程。
 b) 什么是 $L(G)$ 。
 c) 构造一个上下文无关文法用来生成 $L(G)$ 。
 7. 设语言 L 是 $\{a^i b^{2i} a^i \mid i > 0\}$ 。
 a) 使用上下文无关文法的泵引理来证明语言 L 不是上下文无关的。
 b) 构造一个能够生成语言 L 的上下文有关文法 G 。
 c) 给出文法 G 中 $aabbbbbaa$ 的推导。
 d) 构造一个接收 L 的 LBA M 。
 e) 当输入字符串是 $aabbbbbaa$ 的时候, 写出 M 的运算过程。
 *8. 设 L 是 $\{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ 。
 a) L 不是上下文无关的。能够使用上下文无关文法的泵引理对其进行证明吗? 如果能, 则证明之, 否则, 证明泵引理不能够确定 L 不是一个上下文无关的。
 b) 给出一个能够生成 L 的上下文有关文法。
 9. 设 M 是一个字母表为 Σ 的 LBA。略述一个通用的方法来构造用于模拟 M 运算的单调规则集。文法的规则需要包含下面集合中的变量:

$$\{[a_i, x], [a_i, \langle x \rangle], [a_i, x], [a_i, \langle x \rangle], [a_i, q_k x], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, q_k x], [a_i, xq_k], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, \langle xq_k \rangle]\},$$

 其中 $a_i \in \Sigma$, $x \in \Gamma$ 且 $q_i \in Q$ 。这补全了定理 10.3.3 中文法的构造。
 [340] *10. 设 $u \rightarrow v$ 是一条单调规则, 构造一系列能够定义与 $u \rightarrow v$ 一样的字符串的转换的单调规则, 其中每一条规则的右半边有长度最多为 2 的字符串。
 11. 定义一系列的上下文有关规则 $uAv \rightarrow uvwv$, 这些规则能够定义出如同单调规则 $AB \rightarrow CD$ 一样的字符串的转换。提示: 这一系列规则包含三条规则就足够了, 其中每一条的左半边和右半边的字符串长度均为 2。
 12. 使用第 10 题和第 11 题的结果, 来证明每一个上下文有关语言都被这样一个文法生成, 这个文法中, 每一条规则都具有 $uAv \rightarrow uvwv$ 的形式, 其中 $w \in (V \cup \Sigma)^+$ 且 $u, v \in (V \cup \Sigma)^*$ 。
 *13. 设 T 是一个满二叉树。 T 中的一条路是一系列的左下 (L)、右下 (R) 或者上升 (U) 操作。因此一条路径能够用 $\Sigma = \{L, R, U\}$ 上的字符串来标识。考虑语言 $L = \{w \in \Sigma^+ \mid w \text{ 描述了从根开始的最后又返回根的一条路径}\}$ 。例如, λ 、 LU 和 $LRUULU \in L$ 并且 U 、 $LRU \notin L$ 。试确立 L 在乔姆斯基层次中的位置。
 14. 证明上下文有关语言在任意同态下不是封闭的。如果 $h(u) = \lambda$ 表明 $u = \lambda$, 那么说明一个同态是 λ 无关的。证明上下文有关文法在 λ 无关同态运算下是封闭的。
 *15. 设 L 是 Σ^+ 上的一个递归可枚举语言, 且 c 是终结符号, $c \notin \Sigma$ 。说明存在 $\Sigma \cup \{c\}$ 上的一个上下文有关语言 L' , 且对于每一个 $w \in \Sigma^+$, $w \in L$, 当且仅当对于某些 $i \geq 0$ 有 $w c^i \in L'$ 。
 16. 证明每一个递归可枚举语言是某个上下文有关语言的同态像。提示: 使用练习 15。
 17. 一个文法被称为带删除的上下文有关文法, 如果每一条规则都形如 $uAv \rightarrow urw$, 其中 $A \in V$, 且 $u, v, w \in (V \cup \Sigma)^*$ 。证明这一类文法可生成递归可枚举语言。
 18. 一个线性有界自动机是确定型的, 如果对于每一个状态以及带符号, 最多有一个状态转换。证明每一个上下文无关语言都能够被一个确定型线性有界自动机所接收。

19. 设 L 是一个上下文有关语言, 且能够被一个确定型线性有界自动机所接收。证明 \bar{L} 是上下文有关的。要记住, 在一个任意的确定型线性有界自动机中的计算是不需要停机的。

参考文献注释

乔姆斯基层次是由乔姆斯基 [1956, 1959] 提出的。这篇文章包含了无限制文法能够产生递归可枚举语言的证明。线性有界自动机由 Myhill [1960] 提出。线性有界自动机与上下文有关语言之间的关系是由 Landweber [1963] 和 Kuroda [1964] 发展的。练习 10、练习 11 和练习 12 的解决方法, 能够从 Kuroda [1964] 中找到, 这些方案展示了单调性和上下文有关文法之间的关系。

[341]

[342]

第 11 章 判定问题与丘奇—图灵论题

在前几章中，我们利用图灵机来进行字符串的模式识别，识别各种语言，完成函数的计算。然而，很多有意义的问题是比字符串的识别和处理高一个层次的一例如，我们可能对如下这类的问题的答案感兴趣：“这个自然数是否为一个完全平方数？”，或者“一个图中是否存在环路？”，又或者“图灵机执行该运算是否会在 20 次转换之内停机？”。所有这一类的问题都描述了一个判定问题。

严格来说，一个判定问题 P (decision problem P) 是指一系列答案为“是”或者为“否”的问题判定一个自然数是否为一个完全平方数包含了如下的问题：

p_0 : 0 是否为一个完全平方数？

p_1 : 1 是否为一个完全平方数？

p_2 : 2 是否为一个完全平方数？

⋮

每个单独的问题都是该判定问题的一个实例。判定问题 P 的解决方法是指对判定问题的每个实例都能给出正确答案的算法。如果一个判定问题存在解决方法，那么该问题是可判定的 (decidable)。

343

既然对于判定问题的解决方法是一个算法，那么接下来有必要对算法计算这个直观的概念进行简要的回顾。我们还没有定义，而且也没有办法精确地定义译法 (algorithm) 这个概念。这个概念就属于“我无法描述它，但是当我看到的时候，我能知道它是它”的概念类型。但是，我们可以列出算法这个概念所具有的一些基本属性。能够解决判定问题的算法应该具有以下特征：

- 完整性：它应该能够对每个问题的实例产生正确的结果。
- 机械性：它应该包含一个有限的指令序列，而且每条指令的执行都不需要任何的洞察力、创造性和猜测。
- 确定性：面对同样的输入，它应该每次都执行同样的计算。

如果某过程能够满足上述的属性，那么我们就称它为有效的。

标准的图灵机的计算通常是机械的并且是确定的。面对每个输入字符串都能够停机的图灵机是完整的。因为图灵机能够有效地执行计算，我们将使用它作为解决判定问题的框架。我们可以将问题的实例转换成图灵机的输入字符串来构成对判定问题的表述。如果一个输入字符串被接收了，那么与其相对应的问题实例的答案就是肯定的，反之，问题的答案就是否定的。

丘奇—图灵论题证明了：对于每个可以通过有效过程解决的判定问题，都可以通过设计图灵机来解决。关于丘奇—图灵论题的一个更一般性的解释是这样的：任何可以通过算法表述的计算过程都可以通过设计图灵机来实现。本章首先在不同的判定问题、图灵机和递归语言之间建立联系。本章其他部分提出了丘奇—图灵论题，并且讨论了此论断的重要性和影响。

11.1 判定问题的描述

利用图灵机来解决判定问题的第一步是将问题表示成可接收的字符串。这需要为问题构造一个合适的描述。读者可以回忆一下在第 5 章开头所描述的报纸自动贩卖机。要打开贩卖机上面的锁，需要 30 美分的硬币（由面值为 5 美分、10 美分和 25 美分的硬币组成）。如果输入的金额超过了 30 美分，贩卖机就会保留所有的输入。为了防止一个吝啬鬼想要购买报纸但却拒绝支付所必需的最少金额，我们需要提供一个有效的过程来判断输入的各种硬币的总金额是否达到了 30 美分。

为了利用图灵机解决吝啬鬼问题，必须将用自然语言描述的问题实例表示成一个等价的字符串接收问题。我们可以将一组硬币表示成 n, d, q ，这里 n 表示的是面值为 5 美分的硬币， d 表示的是面值为 10 美分的硬币， q 表示的是面值为 25 美分的硬币。利用这种表示方式，我们可以利用图灵机来

对奇高鬼问题的输入进行判断,从而接收 $qnnn$ 、 $nddnd$ 这样的输入,并且拒绝 $nnnd$ 和 $qdqddqq$ 这样的输入。在练习 1 中,读者可以自己构造图灵机来解决这个问题。

图 11-1 给出了利用图灵机来构造判定问题解决方法的两步骤。第一步是选择字母表,并且为问题的实例提供字符串形式的表示。问题表示的性质对于接下来图灵机的设计很有帮助。下面我们将通过“判定一个自然数是否为偶数”这个问题,来对问题表示方式所带来的影响进行解释。表示自然数通常有两种方法:一元表示和二进制表示。采用一元表示的字母表为 $\{1\}$,而数字 n 则表示成 1^{n+1} 。如果采用二进制来进行表示,那么字母表是 $\{0,1\}$ 。

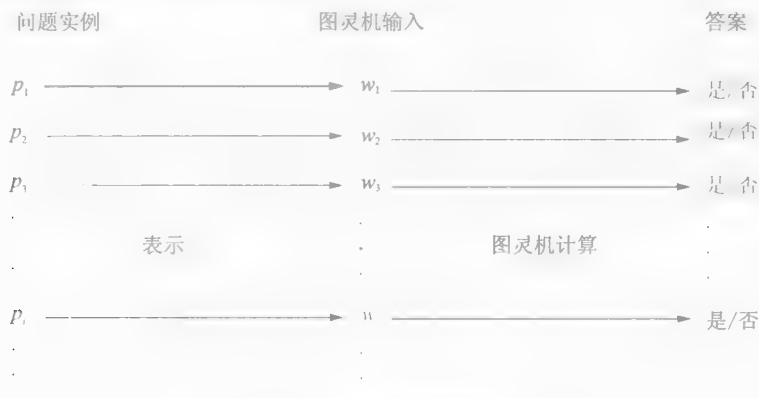


图 11-1 判定问题的解决方法

图灵机采用一元法来解决偶数问题。状态 q_1 和 q_2 分别记录了已经处理的 1 是偶数个还是奇数个。在一元表示法里面,长度为奇数的字符串代表了一个偶数。因此, M_1 表示的语言为 $\{1^i \mid i \text{ 为奇数}\}$ 。



如果利用二进制表示,那么一个偶数的最右边一位为 0,图灵机也就能够接收这样的输入字符串。 M_1 和 M_2 策略的不同解释了图灵机对于不同表示方式的依赖。

将判定问题表示为字符串有很多不同的方式。如果至少有一种表示方法和图灵机的组合能够解决一个判定问题,那么该问题就有一个图灵机的解决方法。当然,一个判定问题也可能有很多种表示方法和图灵机的组合来解决它。

11.2 判定问题和递归语言

我们已经选择标准图灵机作为解决判定问题的形式化工具。一旦我们选定了用来描述问题实例的字符串,那么接下来需要做的就是利用图灵机来对输入的字符串进行分析。既然完整性要求图灵机对每个输入的字符串都能够停机,那么该图灵机能够接收的语言应该是递归的。因此,每个利用图灵机来解决的判定问题都定义了一种递归语言。相应地,每个递归语言 L 也可以被视为某个判定问题的解决方法。因此,判定问题就可以被视为语言的成员问题,即对于字母表 Σ 上的字符串 w ,“ w 是否存在于 L 中”。

借助于判定问题和递归语言之间的关联,我们可以拓展用于检测判定问题的可判定性的技术。既然确定型多道和多带机器的计算能够在标准图灵机上进行模拟,那么利用这类机器也可以给出一个问题的可判定性。

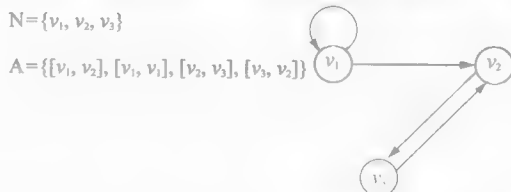
例 11.2.1 “一个自然数是否为一个完全平方数”这个问题是可判定的。在前面例 8.6.2 中的三带图灵机通过将自然数 n 表示为 a^n 解决了这个问题。□

确定型是算法的一个基础性质。然而,通常情况下,如果要接收一个语言,设计一个非确定的图灵机要比设计一个确定型的图灵机容易。在 8.7 节中,我们已经证明了任何可以被非确定型图灵机接收的语言同样可以被确定型的图灵机所接收。而判定问题的解决方法不仅仅要求能够接收一种语言,而且还要求对所有的计算都能够停机。一个对所有计算都停机的非确定型图灵机可以用来构造一个判定的过程。而这些图灵机所接收的语言都是递归的(例 8.23),从而保证存在一个完整的确定的解决方法。

[346]

例 11.2.2 我们利用非确定性来表明问题“有向图中是否存在从节点 v_i 到节点 v_j 的路径”是可判定的。一个有向图包含了一组节点 $N = \{v_1, \dots, v_n\}$ 和边 $A \subseteq N \times N$ 。我们使用 $\{0, 1\}$ 来作为表示图的字母表。节点 v_i 表示为 1^{i+1} , 而有向边 $[v_i, v_j]$ 表示为 $en(v_i)0en(v_j)$, 其中 $en(v_i)$ 和 $en(v_j)$ 分别是节点 v_i 和节点 v_j 的编码。字符串 00 被用来分隔各个有向边。

对于图灵机的输入包括了利用上述方法对于节点和有向边的编码。那么,有向图



可以表示为 110111001101100111011110011110111。这里使用 000 来分隔要判定的问题和图灵机的表示,判定图中是否存在 v_i 到 v_i 的路径的输入为: 11011100110110011101111001111011100011110111。

我们使用非确定的双带图灵机 M 来解决路径问题。图灵机 M 的动作如下:

1. 首先对输入进行检查来确定是否输入的格式代表了一个有向图以及两个用来计算路径的节点。如果格式不满足要求,那么图灵机 M 停机,并且拒绝字符串。

2. 如果图灵机的输入为 $R(G)000en(v_i)0en(v_j)$, 这里 $R(G)$ 是有向图 G 的表示,那么如果 $v_i = v_j$, 图灵机在接收状态停机。

3. 将节点 v_i 的编码添加 0 写在 2 号带上。

4. 假设 v_i 是 2 号带上最右边的节点。从 $R(G)$ 中随机地选取一条从 v_i 到 v_i 的有向边。如果这样的有向边不存在或者 v_i 已经在 2 号带上进行了编码,那么图灵机在拒绝状态停机。

5. 如果 $v_i = v_j$, 那么图灵机在接收状态停机。否则,将 $en(v_i)0$ 写入到 2 号带的尾端,并重复第 4 步的计算步骤。

步骤 4 和 5 能够在 2 号带上产生以节点 v_i 开始的路径。因为步骤 4 保证了只有无环的路径才能写在 2 号带上,所以图灵机 M 的所有计算都会停机。同样,我们可以知道 $L(M)$ 是递归的,而且问题也是可以判定的。□

可以通过描述实例和为了得到肯定的答案而必须满足的条件来定义一个判定问题。使用这种问题定义的方法,我们可以将例 11.2.2 表述成:

有向图的路径问题

输入: 有向图 $G = (N, A)$, 节点 $v_i, v_j \in N$

输出: 是,如果在 G 中存在一条从 v_i 到 v_j 的路径

否,其他情况

[347]

既然可解决的判定问题和递归语言之间存在着这样的对应关系,那我们应该将其称之为问题还是语言呢?在这里,我们约定:当使用一个较高层次的概念来描述问题的时候,我们使用判定问题。当利用字符串的接收来描述问题,并且将问题实例的表述转化为字符串的时候,我们使用递归语言来描述。无论使用哪种表示,如果存在算法能够对问题的每个实例给出正确的回答,或者对每个字符串的成员资格给出正确的判断,那么这个问题是可判定的。

11.3 问题归约

归约是一种解决问题的技术。对于一个新问题，归约技术可以用来避免“重头再来”。归约的目标是将新问题的实例转换为我们了解并已解决的问题形式。归约技术是我们给出问题可判定性的一个有效的工具。同时，在第 12 章中我们还会看到，可以利用归约来证明有些问题是没有算法可以解决的。

定义 11.3.1 假设 L 是定义在 Σ_1 上的语言，而 Q 是定义在 Σ_2 上的语言。如果存在一个图灵可计算的函数 $r: \Sigma_1^* \rightarrow \Sigma_2^*$ 使得 $w \in L$ 当且仅当 $r(w) \in Q$ ，那么我们称 L 可以**多对一归约** (many-to-one-reducible) 到 Q 。

如果一个语言 L 可以通过函数 r 被归约到可判定的语言 Q ，那么 L 是可判定的。假设 R 是执行归约的图灵机，而 M 是接收 Q 的图灵机。那么将 Σ_1^* 上的字符串在 R 和 M 上顺序执行就可以判定 L 的成员问题。



读者应该注意到，用于归约的图灵机 R 既不能判定 L 的成员资格也不能判定 Q 的成员资格；它仅仅是将字符串从 Σ_1^* 转换到 Σ_2^* 。 Q 中的成员资格可以通过 M 来判定，而 L 的成员资格是由 R 和 M 联合在一起判定的。

为了解释从一个语言到另外一个语言的转化，我们将 $L = \{x^i y^j z^k \mid i \geq 0, k \geq 0\}$ 转化为 $Q = \{a^i b^j \mid i \geq 0\}$ 。转化的规则在下面的表中给出。

[348]

归约	输入	条件
$L = \{x^i y^j z^k \mid i \geq 0, k \geq 0\}$	$w \in \{x, y, z\}^*$	$w \in L$
到	$\downarrow r$	当且仅当
$Q = \{a^i b^j \mid i \geq 0\}$	$v \in \{a, b\}^*$	$r(w) \in Q$

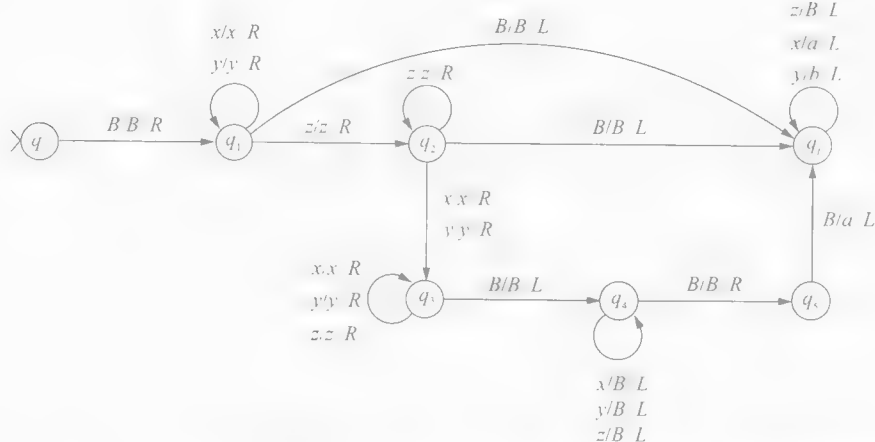
我们将字符串 $w \in \{x, y, z\}^*$ 按照如下的步骤转化为 $r(w) \in \{a, b\}^*$ ：

- 如果在一个 z 字符之后 w 中没有字符 x 和字符 y 出现，则将每个 x 替换为 a ，将每个 y 替换为 b ，并且将 z 擦除。
- 如果在一个 z 字符之后 w 中还有字符 x 和字符 y 出现，则将整个字符串擦除，并在输入的位置写上一个 a 。

接下来的这个表格给出了 Σ_1^* 中一些字符串的转化后的结果。

$w \in \Sigma_1^*$	是否在 L 中	$r(w) \in \Sigma_2^*$	是否在 Q 中
xyy	是	$aabb$	是
$xyyzzz$	是	$aabb$	是
$yxyz$	否	$baab$	否
$xxzyy$	否	a	否
$zyzx$	否	a	否
λ	是	λ	是
zzz	是	λ	是

上面的例子解释了为什么我们将归约称为多对一的转换，在 Σ_1^* 中的多个字符串可能映射到 Σ_2^* 上的同一个字符串。



将 L 归约到 $Q = (x \cup y)^* z^*$ 形式的字符串将在状态 q_1 和状态 q_2 被识别出来, 并转换到状态 q_5 . 而在 z 之后有 x 或者 y 的字符串都将在状态 q_3 被擦除, 并且在带上写上 a 并转化到 q_5 .

例 11.3.1 下面来考虑语言 $L = \{uu \mid u = a^i b^j c^k, i \geq 0\}$ 中的字符串接收问题. 在例 8.2.2 中的图灵机能够接收 $a^i b^j c^k, i \geq 0$. 我们这里将 L 的成员问题归约到 $a^i b^j c^k$ 单个实例的识别问题, 这样就可以通过将归约步骤和例 8.2.2 中的图灵机 M 组合起来加以解决原来的问题. 归约的步骤如下:

1. 拷贝输入的字符串 w . 利用拷贝得到的字符串来确定是否存在字符串 $u \in \{a, b, c\}^*$ 满足 $w = uu$.
2. 如果 $w \neq uu$, 那么将带进行擦除, 并且在输入的位置写上一个 a .
3. 如果 $w = uu$, 那么将拷贝的字符串和输入字符串中的第二个 u 擦除, 而仅在输入位置保留第一个 u .

如果输入的字符串 w 是 uu 格式的, 那么 $w \in L$ 当且仅当 $u = a^i b^j c^k$. 归约过程中并不检查字符 a 、 b 、 c 的个数和先后顺序, 这些都是由例 8.2.2 中的图灵机来完成的.

如果字符串 w 不具有 uu 格式, 那么归约过程就会产生一个字符串 a . 这个字符串接下来将会被 8.2.2 中的图灵机 M 拒绝, 从而表明字符串 w 不在 L 中. \square

如果能够将问题 P 的问题实例转化为问题 Q 的实例, 同时还能保证答案的正确性, 那么我们称一个判定问题 P 可以通过多对一归约到问题 Q . 换一个角度来说, 归约就是转化表述问题实例的字符串. 通常情况下, 如果我们能够在字符串级别上完成这种转换, 那么就可以直接根据问题实例来定义归约. 对于这种技术以及字符串表示形式所带来的影响, 我们将在接下来的例子中加以说明.

例 11.3.2 我们接下来描述在例 11.2.2 中引入的有向图中的路径问题, 它可以被归约为:

确定节点的环问题 (CFN)

输入: 有向图 $G = (N, A)$, 节点 $v_k \in N$

输出: 是, 如果在 G 中有一个环包含 v_k

否; 其他情况。

归约过程需要从 G 构造 G' , 使得在 G 中包含一条从 v_i 到 v_j 的路径就等同于在 G' 中有一个包含的 v_k 环. 归约的第一步是为路径问题的起始点 v_i 找到在 CFN 问题中的 v_k . 当选择了 v_i 作为 CFN 问题中的节点 v_k 之后, 归约过程就可以表示为:

归约	实例	条件
Path 问题 到	图 G , 节点 v_i, v_j	G 中有一个 v_i 到 v_j 的路径,
CFN 问题	图 G' , 节点 v_i	当且仅当 G' 中有一个包含 v_i 的环

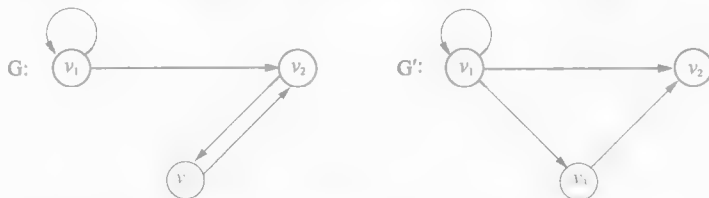
图 G' 可以通过修改图 G 而得到:

- i) 删除所有以 v_i 为目标节点的有向边 $[v_j, v_i]$ 。
- ii) 加入一条边 $[v_j, v_i]$ 。

如果在图 G 中有 v_i 到 v_i 的路径, 那么必然存在一条以 v_i 为起点且只经过一次 v_i 的路径。因此, 删除有向边 $[v_i, v_i]$ 并不会影响是否存在从 v_i 到 v_i 的路径。在第一步删除了所有以 v_i 为目标节点的有向边之后, 图 G 中就不会有包含 v_i 的环, 因为已经没有以 v_i 为目标节点的有向边了。

在第二步加入有向边 $[v_i, v_i]$ 带来的影响是: 在原图 G 中存在一条从 v_i 到 v_i 的路径当且仅当在图 G' 中存在包含 v_i 的环。这样我们就将有向图中的路径问题修改为 CFN 问题了。

将例 11.2.2 中路径问题的实例 G 、 v_3 和 v_i 进行归约, 可以得到:



既然在 G 中没有从 v_3 到 v_1 路径, 那么在 G' 中就没有包含 v_3 的环。

在图灵机的层次上, CFN 问题的实例包含了图 G' 和节点 v_i , 并且可以被表示为 $R(G')000en(v_i)$ 。为了将路径问题中的 G 、 v_3 和 v_i 实例归约为 CFN 问题中的 G' 和 v_i , 我们要做如下的转换:

$$R(G)000en(v_3)en(v_i) = 1101110011011001110111100111101110001111011$$

$$\text{到 } R(G')000en(v_3) = 110111001101100111011110011110110001111$$

执行归约的图灵机应该从输入中删除所有进入 v_i 的有向边, 增加从 v_i 到 v_i 的有向边, 并且从输入的末尾擦除 v_j 。□

351

将判定问题 P 归约到可判定的判定问题 Q 表明了问题 P 也是可判定的。通过将归约过程和解决 Q 的算法顺序连接起来就可以得到 P 的解决方法。

11.4 丘奇—图灵论题

算法计算并不一个很新的概念。实际上, 算法这个词最早由 8 世纪阿拉伯的数学家 Abu Ja'far Muhammad ibn Musa al-Khwarizami 提出。在 Musa al-Khwarizami 的著作中, 他提出了一组解决线性问题和二次问题的规则。这本书被认为是最早的代数著作。几个世纪以来, 人们习惯于使用一些按步执行的机械过程来描述计算、处理和数学推断。在 19 世纪早期, 这种不太正式的使用方法开始走向成熟——数学家们开始尝试着准确地定义算法计算的含意、能力以及局限性。

对于计算能力的探索使得一系列执行算法计算的方法和形式化理论开始出现。数学家们取得了很多的成果: 可以通过定义字符串转化规则, 通过对函数进行估计, 通过抽象计算机, 以及通过编写高级语言中的程序来实现高效的过程。这类系统的例子包括:

- 字符串转化: Post 系统 [Post, 1936], Markov 系统 [Markov, 1961], 非限制文法。
- 函数的估计: 部分和 μ 递归函数 [Gödel, 1931; Kleene, 1936], 演算 [Church, 1941]。
- 抽象计算机: 注册机 [Shepherdson, 1963], 图灵机。
- 编程语言: while 程序 [Kfoury et al., 1982], 第 9 章中的 TM。

在最后一类别中列出的 while 程序, 是采用一种很小的编程语言来编写的程序。该语言仅仅包括赋值、条件以及 for 和 while 语句。尽管 while 程序为了便于对程序进行分析而仅包含很少的语句形式, 不过它却和 C、C++ 以及 Java 这类标准的编程语言有着相同的计算能力。

目前, 我们使用图灵机作为解决判定问题的计算框架; 然而, 我们同样可以选择其他的算法系统。但是这样做, 会不会影响我们解决问题的能力呢? 理想的答案应该是否定的——问题解决方法的存在与否应该是问题自身的特征, 而不应该是我们选择的算法系统所产生的制品。丘奇—图灵论题验

证了这一点。

[352]

那么前面提到的算法系统有什么样的共同点呢？这些算法都可以执行同样的计算。这个结论是值得注意的，因为这些系统在设计的时候是用来对不同类型的数据执行不同类型操作的。然而，读者已经在前文中看到了它们等价的一个例子，并且将在第 13 章中看到另外一个例子。在 10.1 节中，我们证明了图灵机的计算可以通过非限定文法来进行模拟。因此，任何由非限定文法所定义的语言都能够被图灵机所接收。在第 13 章中，我们将给出算法方法的定义，并对 Gödel 和 Kleene 提出的数论函数进行评价。这种方法具有和图灵机相同的功能。

很多计算方法所产生的系统都具有相同的计算能力。这个事实让我们相信这些系统能够定义算法计算能力的边界。算法和执行计算的系统是密不可分的。然而，对于这些系统来说，它们的能力有一个明确定义的边界。丘奇—图灵论题对算法计算的能力和限制进行了形式化的表述。我们接下来首先对丘奇—图灵论题进行解释。

判定问题中的丘奇—图灵论题 判定问题存在有效的解决过程当且仅当图灵机能够对问题所有的输入停机并且能够给出正确的答案。

判定问题的解决方法需要对每个问题实例的计算都返回正确的答案。如果放宽了这个限制，那么我们得到的就是判定问题的部分解决方法。一个判定问题 P 的部分解决方法并不具有完整性，但是它能够对每个结果为“是”的判定问题实例 $p \in P$ 做出正确的反应。而如果问题 p 的答案是否定的，那么计算过程或者返回否定的结果，或者不能够产生结果。也就是说，计算仅仅能够识别正确的实例。

正如我们可以将判定问题的解决方案转化为递归语言的成员问题一样，判定问题的部分解决方法同样等同于一个递归可枚举语言的成员问题。丘奇—图灵论题同样包含了识别语言和定义语言的算法。

语言识别问题中的丘奇—图灵论题 判定问题 P 是部分可解决的，当且仅当有图灵机能够接收问题 P 所有结论为“是”的实例。

图灵机使用带上的符号来执行计算的功能，当图灵机停机的时候，带上保留的就是计算的结果。解决判定问题的函数化方法是使用 1 和 0 来分别表示肯定和否定的结果。归约问题答案的方法不会影响到存在图灵机解决方法的问题集合（练习 9.4）。因此，有关可计算函数的丘奇—图灵论题包含并扩展了论题的前两个版本。

[353]

可计算函数中的丘奇—图灵论题 一个函数 f 是可以有效计算的，当且仅当存在图灵机能够执行 f 。

第 13 章，我们在图灵可计算函数与 μ 递归函数之间建立了一致性，我们将会给出丘奇—图灵论题一个更细致的版本，同时给出一个从可计算的数论函数到可计算函数的自然泛化过程。

为了理解丘奇—图灵论题的内容，我们必须理解断言的本质。丘奇—图灵论题并不是一个数学定理，它无法被证明。因为证明它需要对“有效过程”这个直观的概念进行形式化的定义。而这个定义可能是无法给出的。关于这一点，可以通过设计一个无法通过图灵机来计算的有效过程来证实。图灵机和其他的算法系统的等价性，图灵机体系结构的健壮性，以及无法找到一个反例都有力地证明了上面说的过程是无法找到的。

丘奇—图灵论题为证明判定算法的存在提供了一条捷径。我们可以通过描述解决问题的有效步骤来完成证明，而不需要为判定问题建立一个图灵机。丘奇—图灵论题确保了总会有相应的图灵机来解决这个问题。在表现图灵机计算能力的过程中，我们一直在隐式地使用丘奇—图灵论题。对于复杂的机器，我们只是给出图灵机计算动作的简要描述。我们知道，如果需要，整个图灵机总可以被显式地构造出来。

11.5 通用机

20 世纪 40 年代中期计算机设计的一个重要突破就是存储程序的计算模型。早期的计算机被用来执行一个任务。输入可能不同，但是对于每个输入执行的是相同的程序。如果对指令进行修改那么就需要对硬件进行重新配置。在存储程序的模型中，指令和数据是以电子的形式被装载到内存中的。在

存储程序的计算机中, 计算是从内存中提取一条指令并执行它的循环过程。

在前些章节里面提到的图灵机, 就如同早期的计算机一样, 只能执行单个集合的指令。图灵机的体系结构中有自己存储程序的概念, 这比最早的存储程序的计算机领先了十年。通用图灵机 (universal turing machine) 可以被用来模拟任意图灵机 M 的计算过程。为了完成这一点, 通用图灵机的输入必须包含图灵机 M 的表示, 以及要被处理的字符串 w 。为了简单起见, 我们假设 M 是能够接收停机的标准图灵机。通用图灵机的活动可以下面的图来解释:

354



图中的 $R(M)$ 是 M 的表示, 而标明了 loop 的输出表明了 U 的计算不会终止。如果 M 停机并且接收了输入 w , 那么 U 将执行同样的操作。如果 M 对于 w 不停机, 那么 U 也不停机。 U 之所以被称为通用机, 是因为 U 能够模拟任何的图灵机 M 。

构造通用图灵机的第一步是为图灵机设计字符串的表示。因为任意的符号都可以在 $\{0, 1\}$ 上的字符串进行编码, 我们规定通用图灵机的输入字母表为 $\{0, 1\}$, 而带字母表为 $\{0, 1, B\}$ 。通用图灵机的状态命名为 $\{q_0, q_1, \dots, q_n\}$, 其中 q_0 为起始状态。

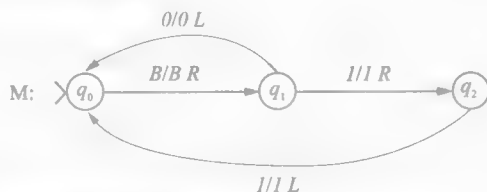
标准图灵机可以通过它的转换函数来定义。标准图灵机的转换格式为 $\delta(q_i, x) = [q_j, y, d]$, 这其中 $q_i, q_j \in Q$; $x, y \in \Gamma, d \in \{L, R\}$ 。我们使用 1 的字符串来对 M 中的元素进行编码。假设 $en(z)$ 代表了符号 z 的编码。那么转换 $\delta(q_i, x) = [q_j, y, d]$ 就可以用字符串编码为:

符号	编码
0	1
1	11
B	111
q_0	1
q_1	11
...	...
q_n	1^{n+1}
L	1
R	11

$$en(q_i)0en(x)0en(q_j)0en(y)0en(d)$$

0 用来分离转换函数中不同的部分。机器的表示由经过编码的不同的转换函数组成的。两个连续的 0 用来分隔不同的转换函数。而表示的开始和结束分别包含三个 0。

例 11.5.1 图灵机的计算过程如图



它对于空串和以 1 开始的字符串都能够停机, 而对于以 0 开始的字符串则不停机。 M 编码后的转换函数在下面的表中给出。

355

那么我们就可以将 M 表示为字符串

000101110110111011001101010101001101101110110110011101101010101000

□

可以构造一个图灵机来判断字符串 $u \in \{0, 1\}^*$ 是否是一个确定型图灵机的表示。检查的步骤首先判断 u 是否包含了 000 作为前缀, 然后是由 00 分开的一系列有限转化序列, 最后以 000 作为结尾。如果字符串满足这些要求, 那么它就是某个图灵机 M 的表示。如果状态和输入符号的结合都是不同的, 那么图灵机 M 就是确定型的。

接下来, 我们来描述一个三带确定型通用图灵机 U 的设计过程。 U 的计算从带 1 上的输入开始。如果输入的字符串格式为 $R(M)w$, 那么就在带 3 上模拟 M 中执行输入为 w 的计算过程。 U 的计算包

转化	编码
$\delta(q_0, B) = [q_1, B, R]$	101110110111011
$\delta(q_1, 0) = [q_0, 0, L]$	1101010101
$\delta(q_1, 1) = [q_2, 1, R]$	110110111011011
$\delta(q_2, 1) = [q_0, 1, L]$	1110110101101

含了如下的动作:

1. 如果输入的字符串的格式不是 $R(M)w$, 这里 M 代表的是确定型图灵机, w 为字符串, 那么 U 将一直向右移动。

2. 如果输入的字符串的格式是 $R(M)w$, 则将输入的字符串 w 写入到带 3 上的开始位置。然后将带头重新放在带上最左边。带 3 的配置就是在 M 上执行输入为 w 的初始配置。

3. 将一个 1 和状态 q_0 的编码写入到带 2 上。

4. 在带 3 上模拟 M 的转化, M 的转换是由在带 3 上的符号以及在带 2 上的状态来决定的。假设 x 是带 3 上的符号, 而 q_i 是带 2 上的状态的编码。

a) 在带 1 寻找满足 $en(q_i)$ 和 $en(x)$ 的转换。如果没有这样的转换, 那么 U 停机接收输入。

b) 如果带 1 上包含转换 $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$ 的编码, 那么

i) 在带 2 上将 $en(q_i)$ 替换为 $en(q_j)$

ii) 将符号 y 写入到带 3 上

iii) 带 3 上的带头按照 d 声明的方向进行移动。

5. 继续步骤 4 中的计算, 从而模拟 M 的下一个转换。

定理 11.5.1 语言 $L_H = \{R(M)w \mid M \text{ 在输入 } w \text{ 上停机}\}$ 是递归可枚举的。

证明: 通用图灵机 U 接收形式为 $R(M)w$ 的输入, 这里 $R(M)$ 是图灵机的表示, 而且 M 在输入为 w 时停机。对于其他的字符串, U 不停机。因此 U 的语言就是 L_H 。 ■

语言 L_H 通常被称为停机问题的语言。如果一个字符串是图灵机 M 的表示和输入字符串 w 的结合, 并且图灵机 M 在输入 w 上停机, 那么该字符串就属于 L_H 。

通用图灵机 U 的计算以 $R(M)w$ 为输入, 并且模拟了 M 在输入 w 的情形。我们可以通过设计复杂的图灵机来模拟一个图灵机的结果。当我们说一个图灵机 M' “在输入 w 的情况下运行了 M ”, 那么是指 M' 接受了 $R(M)$ 和 w , 并且按照通用图灵机的方式模拟了 M 的计算。

例 11.5.2 对于判定问题

第 n 次转化停机问题

输入: 图灵机 M , 字符串 w , 整数 n

输出: 是; 如果在 M 上输入 w , 经过 n 次转化恰好停机

否, 其他情况。

其解决方法可以通过模拟 M 的计算来完成。直观感觉上, 该解决方法应该是“在输入 w 的情况下运行了 M ”, 并且计算 M 的转换次数。

解决此问题的图灵机 U' 可以在通用图灵机上增加第四条带来记录在 M 的计算过程中转换的次数。问题的实例将以 $R(M)w0001^n$ 的字符串形式来进行表示, 这里将用单一进制的 n 和 $R(M)w$ 以三个 0 加以分隔开来。对于输入字符串 u , U' 的计算动作如下:

1. 如果输入的字符串 u 不是以 0001^{n+1} 结尾的, 那么 U' 停机拒绝输入。

2. 将字符串 1^n 写入到带 4 上的起始位置, 将 0001^{n+1} 从带 1 上的字符串擦除; 并将带 4 上的带头指向起始位置。

3. 如果带 1 上的剩余的字符串不具有 $R(M)w$ 的格式, 那么 U' 停机拒绝输入。

4. 将字符串 w 拷贝到带 3 上, 并且将 q_0 的编码写入到带 2 上。

5. 按照通用图灵机的策略, 在带 1 寻找和带 3 上的符号 x 以及带 2 上状态 q_i 的编码匹配的转换

a) 如果没有针对 q_i 和 x 的转换, 而且在带 4 上读入了一个 1, 那么 U' 停机, 拒绝输入。

b) 如果没有针对 q_i 和 x 的转换, 而且在带 4 上读入了一个空格, 那么 U' 停机, 接收输入。

c) 如果在带 1 上有针对 $\delta(q_i, x)$ 编码的转换, 并且在带 4 上读入了一个空格, 那么 U' 停机, 拒绝输入。

d) 如果在带 1 上有针对 $\delta(q_i, x)$ 编码的转换, 并且在带 4 上读入了一个 1, 那么在带 2 和带 3 上对转化进行模拟, 并且将带 4 上的带头向右移动一个单位。

6. 继续第 5 步的计算, 并且检查 M 的下一个转换。

如果 M 在第 n 次转换之前就停机, 那么 $R(M)w0001^{n+1}$ 将会在步骤 5(a) 中被拒绝。在模拟了 M 的 n 次转换之后, 在带 4 上的计数器读入一个空格。如果 M 在这个点上没有可用的转换, 那么 U 接收输入; 否则, 输入将会在步骤 5(c) 中被拒绝。□

11.6 练习

1. 给出能够解决在 11.1 节中描述的各魔鬼问题的图灵机状态图。硬币表示为集合 $\{n, d, q\}$ 的元素, 这里 n 、 d 和 q 分别代表了五分的硬币、十分的硬币和二十五分的硬币。

从练习 2 到练习 7, 我们描述了解决特定判定问题的图灵机。在这里使用例 11.2.2 作为定义图灵机计算动作的模型。读者不需要明确地给出解决方案中的转换函数或者状态图。读者可以在解决方案中使用多带和非确定型图灵机。

2. 设计一个图灵机来判定定义在 $\{0, 1\}$ 上的字符串 u 和 v 是不是相等。在计算开始之前, 带上的字符串为 $BuBvB$, 而且要求在不多于 $3(\text{length}(U) + 1)$ 次转换之内完成判定。

3. 使用单一进制表示法来描述自然数, 设计一个图灵机来判定一个自然数是不是素数。

4. 使用单一进制表示法来描述自然数, 设计一个图灵机来判定“ 2^n ”问题。提示: 输入为自然数 i 的表示, 而如果对于某个 n , 满足 $i = 2^n$, 则输出“是”, 否则输出“否”。

5. 如果一个有向图中包含至少一个环, 那么我们把这个图成为有环的。使用 11.2 节中对于有向图的表示方法, 设计一个图灵机来判定一个有向图是否有环的。

6. 有向图的遍历是指路径 p_0, p_1, \dots, p_n , 而且满足:

i) $p_0 = p_n$;

ii) 对于 $0 < i, j \leq n, i \neq j$ 意味着 $p_i \neq p_j$;

iii) 图中的每个节点都出现在路径中。

也就是说, 一个遍历访问了图中的每个节点一次且仅一次, 并且在起始点结束。设计一个图灵机判定一个有向图中是否包含了一条遍历的路径。要求使用 11.2 节中对于有向图的表示方法。

7. 假设 $G = (V, \Sigma, P, S)$ 是正则文法。

a) 在 $\{0, 1\}$ 上为文法 G 构造表示法。

b) 设计一个图灵机来判定字符串 $w \in \Sigma^+$ 是否属于 $L(G)$ 。可以使用非确定型来帮助设计图灵机。

8. 构造一个能够将语言 L 归约到 Q 的图灵机。在下面的每个题目中, L 使用的字母表都是 $\{x, y\}$, 而 Q 使用的字母表都是 $\{a, b\}$ 。

a) $L = (xy)^*$ $Q = (aa)^*$

b) $L = x^*y^*$ $Q = a^*b$

c) $L = \{x^i y^{i+1} \mid i \geq 0\}$ $Q = \{a^i b^i \mid i \geq 0\}$

d) $L = \{x^i y^j z^i \mid i \geq 0, j \geq 0\}$ $Q = \{a^i b^i \mid i \geq 0\}$

e) $L = \{x^i (yy)^i \mid i \geq 0\}$ $Q = \{a^i b^i \mid i \geq 0\}$

f) $L = \{x^i y^j x^i \mid i \geq 0\}$ $Q = \{a^i b^i \mid i \geq 0\}$

9. 设 M 是图灵机



a) 那么 $L(M)$ 是什么?

b) 利用 11.5 节中的编码方法给出 M 的表示。

10. 构造一个图灵机来判定利用 $\{0, 1\}$ 构造的字符串是否是一个非确定型图灵机的编码。如果需要判定输入是否是一个确定型图灵机的编码, 那么该图灵机需要做什么样的修改?

11. 设计一个图灵机, 接收在 $\{0,1\}$ 上的输入字符串 u , 如果

i) 对于某个图灵机 M 和输入字符串 $u, u = R(M)w$

ii) 当在 M 上输入 w , 能够有转换执行计算, 并且输出一个 1。

[359] 图灵机不需要对所有的输入都停机。

12. 给定一个任意的图灵机 M 和输入字符串 w , 那么 M 能否在少于 100 次的转换中接收 w ? 设计一个图灵机来解决这个判定问题。

13. 证明判定问题

输入: 图灵机 M

输出: 是; 如果 M 从空白带开始计算, 它的第三个转换产生了一个空格
否; 其他情况。

是可以判定的。如果图灵机在第三次转换之前停机, 那么答案就是否。

14. 证明下面的判定问题

输入: 图灵机 M

输出: 是; 如果存在 $w \in \Sigma^*$ 使得 M 的转化超过了 10 次
否; 其他情况。

是可判定的。

15. 在 11.5 节中引入的通用图灵机能够模拟其他图灵机的接收停机的动作。因此, 表示形式 $R(M)$ 并不包含对接收状态的编码。

a) 将对图灵机 M 的表示 $R(M)$ 进行扩展, 使它能够显示地对 M 的接收状态进行编码

b) 设计一个通用图灵机 U , 使得它能够接收 $R(M)w$ 形式的输入当且仅当图灵机 M 在终结状态接收了输入 w 。

参考文献注释

图灵 [1936] 设计了用于理论计算的图灵机, 从而能够执行所有有效的计算。这个论点, 由丘奇 [1936] 总结成为丘奇—图灵论题。图灵在 1936 年的论文中同样引入了通用图灵机的设计。关于存储程序计算机的设计方案最早由 von Neumann [von Neumann, 1945] 提出, 而第一个工作模型在 1949 年出现。

[360] 在我们设计的通用图灵机中, 限制了图灵机的输入和带上可以使用的字母表分别为 $\{0,1\}$ 和 $\{0,1,B\}$ 。任意图灵机都可以利用一个使用上述的字母表的图灵机来模拟, 相关的证明可以参考 Hopcroft 和 Ullman 的工作 [1979]。

第 12 章 不可判定性

丘奇-图灵论题断言图灵机可以解决任何存在有效解决过程的判定问题。图灵机的计算不会受到“真实”计算设备所固有的物理限制。因此，对于图灵机来说，问题是否可以解决取决于问题自身，而不是取决于内存和中央处理器可用的时间。丘奇-图灵论题对于不可判定性也有相应的结论。如果一个判定问题不能够被图灵机所解决，那么也就不存在任何有效的过程能够解决它。如果一个判定问题不存在算法的解决方案，那么我们就称它为不可判定的（Undecidable）。

在 9.5 节中，我们已经证明了图灵机的数目是可数的。然而，在一个非空字母表上的语言的数目却是不可数的。这也就意味着有些语言的成员问题是不可判定的。这两者数目的比较让我们确信了不可判定问题的存在，但是却没有明确地给出不可判定问题的描述。在本章中，我们将给出一些特定的判定问题，并通过这些问题来探讨图灵机计算能力和相关文法的变种。后面我们可以看到，甚至多米诺游戏问题也是不可判定的。

我们考虑的第一个问题是图灵机停机问题。考虑到图灵机停机问题的重要性，我们首先用 C 程序而不是用图灵机来描述它。用 C 程序描述的图灵机停机问题是这样的：

C 程序的停机问题

输入：C 程序 Prog，

针对 Prog 的输入文件 input

输出：是；如果以 input 为输入时 Prog 运行会停止
否；其他情况

361

如果对于 C 程序的停机问题是可判定的，那么编程人员所头疼的问题之一——无限循环问题，将成为过去。程序的执行将成为一个包含两步骤的过程。

1. 运行能够解决停机问题的算法，来判断 Prog 和 input 是否停机。

2. 如果算法表明了 Prog 停机，那么利用 input 运行 Prog。

停机问题的算法并不会告诉我们程序实际运行的结果，而只是告诉我们程序是否会产生结果。在利用停机问题得到了肯定的回复之后，接下来可以通过在 Prog 上运行 input 来得到结果。不幸的是，C 程序的停机问题就像它对应的图灵机停机问题一样，都是不可判定的。

在本章的前 4 节，我们会考虑输入字母表为 $\{0,1\}$ ，带上的字母表为 $\{0,1,B\}$ 的图灵机。字母表上的限制并不会对图灵机的计算能力造成影响，这是因为任意图灵机 M 的计算都可以利用使用上面字母表的图灵机来模拟。模拟的过程仅需要将 M 上的符号使用 $\{0,1\}$ 来进行编码。这一点和数字计算机所采用的编码方法是一致的。数字计算机中一般采用 ASCII、EBCDIC 或者 Unicode 来对字符进行二进制编码。

12.1 图灵机的停机问题

最著名的不可判定问题是和图灵机自身的性质相关的。接下来我们描述图灵机停机问题：给定任意的一个图灵机 M 和输入字母表 Σ ，以及字符串 $w \in \Sigma^*$ ，那么在 M 上运行输入 w ，会不会停机？我们接下来将证明没有算法能够解决图灵机停机问题。图灵机停机问题的不可判定性是计算机科学理论里面的基础性结论之一。

理解问题的描述是很重要的。对于一个特定的图灵机，我们可能能够判定它是否会对任意给定的字符串停机。实际上，在例 8.3.1 中的图灵机仅对包含 aa 子字符串的输入停机。而图灵机停机问题的解决方法则需要能够对每个可能的图灵机和输入字符串实例给出结论。

既然图灵机停机问题涉及到图灵机，那么它的输入必然包含了一个图灵机实例，或者具体地说是

362

一个图灵机实例的表示。我们将使用在 11.5 节中所设计的图灵机表示方法，将输入在 $\{0,1\}^*$ 上的图灵机采用 $\{0,1\}^*$ 进行编码。停机问题的不可判定性的证明并不依赖于特定编码的特征。证明的结论应该对于任何的图灵机编码表示都是有效的。在这之前，我们先将图灵机的表示命名为 $R(M)$ 。

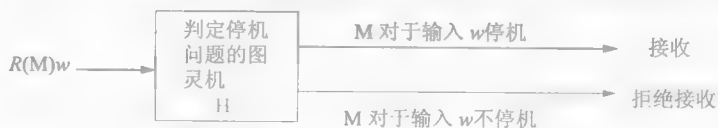
我们将利用反证法来证明停机问题的不可判定性。我们假设图灵机 H 能够解决判定问题。通过对 H 进行简单地修改我们可以得到另外一个图灵机 D ，这样就会造成一个矛盾的现象：当图灵机 D 使用自己的表示作为输入时候，将会出现不可能的情况。既然假设存在能够解决停机问题的 H 会出现矛盾，那么停机问题便是不可判定的。

定理 12.1.1 图灵机的停机问题是不可判定的。

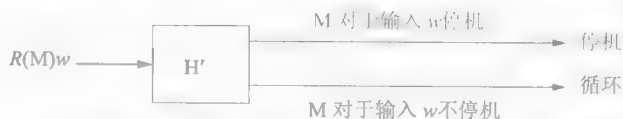
证明：假设图灵机 H 能够解决停机问题。那么 H 能够接受字符串 $z \in \{0,1\}^*$ ，如果

- i) z 包含了一个图灵机 M 的表示，并且后面接着一个字符串 w ；
- ii) M 对于以 w 为输入的计算会停机。

如果上述的两个条件有一个不满足，则 H 拒绝接收输入。图灵机 H 的操作情况如下图所示：



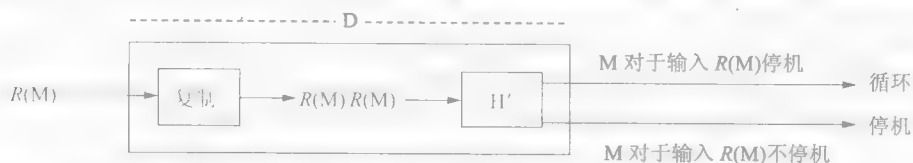
下面我们将 H 进行修改从而得到一个新的图灵机 H' 。 H' 的计算和 H 大致相同，只是当 H 进入接收停机状态的时候， H' 会继续计算。这时候， H' 将一直向右移动。 H' 的转换函数是在 H 的转换函数的基础上增加了，当 H 进入接收状态时， H' 将无限制的向右移动。 H' 的行为通过下面的图来表示。



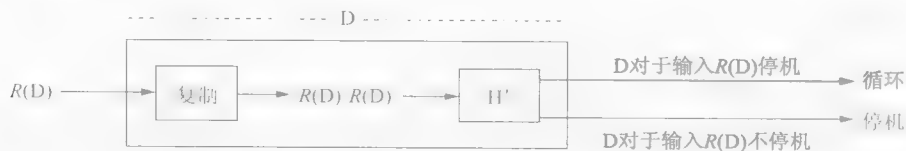
在证明中，我们只关心计算是会停止还是会不停地继续。后一种情况在图中采用循环来表示。

363

图灵机 H' 将和一个负责复制的图灵机共同构造图灵机 D 。 D 的输入是图灵机的表示 $R(M)$ 。 D 的计算从根据输入 $R(M)$ 来产生 $R(M)R(M)$ 开始。接下来的计算包括了在 H' 上运行 $R(M)R(M)$ 。



D 的输入可能是建立在字母表 $\{0,1,B\}$ 上的任意图灵机的表示。而且， D 本身也是这样的图灵机。那么接下来，我们考虑使用 D 来计算 $R(D)$ 。我们将前面图中的 M 替换为 D 、 $R(M)$ 替换为 $R(D)$ ，那么就可以得到：



观察这张图，我们会发现 D 在输入 $R(D)$ 上停机，当且仅当 D 在 $R(D)$ 上不停机。这显然是不可能的。然而， D 是从能够解决停机问题的图灵机 H 直接构造得来的。这个假设直接造成了前面的矛盾。因此，我们总结得出图灵机停机问题是不可判定的。■

在前面的证明中，我们使用了自引用和对角线化。为了得到对角线化证明过程中一个标准的关系

表, 我们可以考虑每个代表图灵机的字符串 $v \in \{0, 1\}^*$ 。如果 v 不具有 $R(M)$ 的格式, 那么我们将 v 赋给一个只有一个状态的而且无转换的图灵机。因此, 我们可以针对字符串 $\lambda, 0, 1, 00, 01, \dots$, 排列图灵机 $M_0, M_1, M_2, M_3, M_4, \dots$ 。那么我们考虑在水平和垂直的轴上都列出了图灵机的一张表表中的第 i, j 项是

$$\begin{cases} 1, & \text{如果 } M_i \text{ 运行 } R(M_j) \text{ 时停机} \\ 0, & \text{如果 } M_i \text{ 运行 } R(M_j) \text{ 时不停机。} \end{cases}$$

表的对角线代表了自引用问题的答案。“ M_i 运行自己的表示会不会停机?” 为了说明这个问题, 我们构造 D 产生了矛盾的情况。

同样, 可以对任意输入字母表的图灵机建立停机问题不可判定的证明。本方法的一个根本特征是它能够利用自身的输入字母表来将图灵机编码为字符串。为了构造这种编码, 使用两个符号就足够了。

停机问题的不可判定性和通用图灵机模拟图灵机计算的能力结合在一起, 证明了递归语言是递归可枚举语言的一个真子集。推论 12.1.2 则是利用递归语言来对停机问题的不可判定性进行了另一种描述。

推论 12.1.2 在 $\{0, 1\}$ 上的语言 $L_H = \{R(M)w \mid R(M) \text{ 图灵机 } M \text{ 的表示, 而 } M \text{ 对 } w \text{ 停机}\}$ 不是递归语言。

推论 12.1.3 递归语言是递归可枚举语言的真子集。

证明: 通用图灵机 U 接收 L_H 字符串能够被 U 接收仅当它的形式为 $R(M)w$, 而且在 M 上运行输入 w 时, M 停机。 L_H 能够被图灵机 U 接收表明了 L_H 是递归可枚举的, 但是推论 12.1.2 表明了 L_H 不是递归的。■

在练习 8.26 中, 我们证明了如果 L 和 L 都是递归可枚举的, 那么 L 是递归的。我们将它和推论 12.1.2 结合在一起得到推论 12.1.4。

推论 12.1.4 语言 $\overline{L_H}$ 不是递归可枚举的。

推论 12.1.4 表明了没有算法能够接收或识别语言 $\overline{L_H}$ 中的字符串。从模式识别的角度来看, 我们可以使用机器来识别具有一些共有模式的字符串。当一个语言不是递归可枚举的时候, 语言成分中任何共有的模式都过于复杂, 以至于无法通过算法来发现。

364

365

12.2 问题归约和不可判定性

我们在第 11 章中引入了归约, 并把它作为构造判定问题解决方法的有力工具。如果存在一个图灵机可计算的函数 r 能够将判定问题 P 的实例转换为判定问题 Q 的实例, 并且保持了对于问题 P 实例的答案, 那么

归约	输入	条件
P	实例 p_0, p_1, \dots	p_i 的答案是肯定的
到	$\downarrow r$	当且仅当
Q	实例 q_0, q_1, \dots	$r(p_i)$ 的答案是肯定的

那么我们说 P 可以归约到 Q 。在第 11 章中, 我们使用表格的形式来对从 P 到 Q 的归约进行描述。归约对于不可判定性的证明同样有着重要的作用。如果 P 是不可判定的, 并且可以归约到 Q , 那么 Q 必然也是不可判定的。这是因为如果 Q 是可判定的, 那么将从 P 到 Q 的归约过程加上 Q 的解决方法, 将会产生 P 的解决方法: 对于 P 的输入 p_i

i) 使用归约将 p_i 转化到 $r(p_i)$ 。

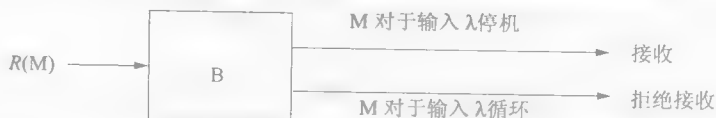
ii) 使用 Q 的解决方法来判定 $r(p_i)$ 的答案。

因为 r 是一个归约的过程, 那么判定问题 P 对于输入 p_i 的结果和 Q 对于 $r(p_i)$ 的结果是一致的。顺序执行归约和解决 Q 的算法能够得到 P 的解决方法。这和已知 P 是不可判定的是相互矛盾的。因此, 关于 Q 是可判定的假设无法成立。

空白带问题 (blank tape problem) 是判定图灵机是否能够在空白带上开始计算并最终停机的问題。空白带问题是图灵机停机问题的特例, 因为它仅关注当输入为空时的停机问题。我们下面将会把图灵机停机问题归约到空白带问题, 这样我们就证明了空白带问题是不可判定的。

定理 12.2.1 没有算法能够判定一个任意的从空白带开始计算的图灵机是否能够停机

证明：假设存在图灵机 B 能够解决空白带问题，那么我们可以将它表示为

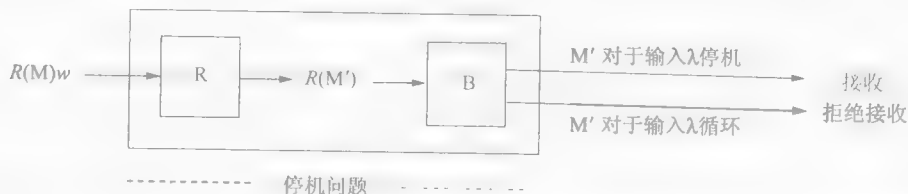


将图灵机停机问题归约到空白带问题是由图灵机 R 实现的。图灵机 R 的输入是图灵机 M 的编码表示, 接下来是输入字符串 w 。 R 计算的结果是图灵机 M' 的编码表示。图灵机 M' 的动作作为:

1. 将 w 写在空白带上,
2. 将带头放到初始位置并将机器的状态设为 M 的初始状态,
3. 运行 M 。

$R(M')$ 是通过在 $R(M)$ 上增加编码的转换, 并对 M 的初始状态进行重命名而得到的。图灵机 M' 在空白带上运行停机当且仅当在输入 w 时候 M 停机。

[366] 可以得到下面图中的图灵机。



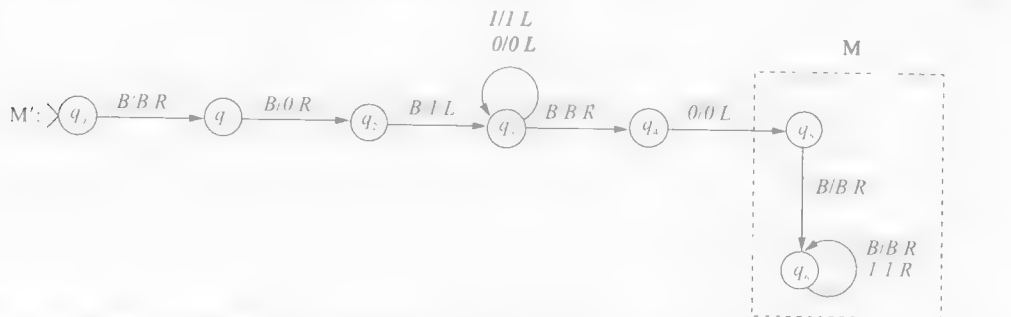
追踪整个计算的过程, 我们可以看到组合的机器提供了停机问题的解决方法。因为预处理器 R 可以将停机问题归约到空白带问题, 所以空白带问题是不可判定的。

将停机问题归约到空白带问题的预处理器 R，修改了 M 的表示从而构造了图灵机 M' 的表示。例 12.2.1 给出了预处理器 R 执行转换的结果。

例 12.2.1 M 是对任何包含 0 的输入字符串停机的图灵机。对于 M 的编码 $R(M)$ 是

0001011101101110110011011101101110110011011011011011000

输入 $R(M)01$, 预处理器 R 构造了图灵机 M' 的编码。



当使用空白带的时候, M' 的最初五个状态用来在输入位置上写入 01. M 的拷贝接下来在 $B01B$ 带上运行. 从构造的过程中我们可以看到, M 在输入 01 时停机当且仅当 M' 从空白带开始运行停机. \square

因为空白带问题是停机问题的子问题，那么接下来我们考虑问题、子问题和不可判定性之间的关系。下面的问题都是对停机问题的输入进行限定而得的。

子问题	输入	可判定
空白带问题	$R(M)$, (输入固定)	不可判定
通用图灵机停机问题 U	(机器固定), $R(M)w$	不可判定
例 8.3.1 的 M 停机问题	(机器固定), w	可判定

对于通用图灵机停机问题是判定对于输入 $R(M)w$, 通用图灵机 U 是否会停机。这个问题的解决方法将可以用来判定任意的图灵机 M 对于输入 w 是否会停机, 这样就可以得到图灵机停机问题的解决方法。从前面的表中我们可以看出: 一个不可判定问题的子问题是否是可判定的, 取决于子问题对于问题特征的保留程度。另一方面, 如果 Q 是 P 的子问题, 并且 Q 是不可判定的, 那么 P 必然是不可判定的。这是因为任何能够解决问题 P 的算法都可以解决它的子问题。

将停机问题归约到空白带问题是由图灵可计算函数 r 完成的, r 将 $R(M)w$ 转换为 $R(M')$ 。定理 12.2.1 和例 12.2.1 表明了如何将图灵机的表示 $R(M)$ 修改为 $R(M')$ 。在接下来的例子中, 我们将对归约给出一个较高层次的解释, 并省略掉字符串表示的操作细节。

12.3 其他的停机问题

我们已经证明了图灵机停机问题和空白带问题都是不可判定的。对于图灵机, 我们还有很多其他的问题, “计算过程中图灵机是否会进入一个特定的状态”, 或者“计算是否会在它的终结状态上写下一个特定的符号”等等。利用图灵机停机问题的不可判定性, 我们可以对很多这样的问题进行归约, 从而得出这些问题也是不可判定的。

下面, 我们将利用图灵机计算初始状态重入的问题来展示证明此类问题不可判定性的大致策略。初始状态重入的计算满足 $q_0 B w B \vdash u q_0 v B$ 。计算不需要在终结状态停机或者根本不需要停机, 所需要的只是机器在计算开始后的某个点能够返回它的初始状态 q_0 。

我们通过将图灵机停机问题归约到初始状态重入问题来证明它是不可判定的。归约的形式正如表 12.3.1 所示, 对于图灵机停机问题中的 M 和初始状态重入问题中的 M' , 我们将使用相同的字符串 w 。

子问题	输入	可判定
空白带问题	图灵机 M , 字符串 w	M 在 w 下停机
到	↓	当且仅当
重入问题	图灵机 M' , 字符串 w	M' 运行 w 时进入了初始状态

设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, 而 w 是图灵机停机问题实例中的输入字符串。我们需要构造一个图灵机 M' , 使得 M' 满足: 当输入为 w 时, M' 重新进入初始状态当且仅当图灵机 M 在输入 w 时会停机。首先需要注意到, 在任意一个图灵机中, 我们无法直观地将图灵机停机问题和初始状态重入问题联系在一起。在归约过程的设计中, 我们需要将两者联系起来。

构造 M' 的思路是: 从 M 开始, 增加一个具有和 q_0 具有相同转换函数的状态 q'_0 , 并且为 q'_0 增加向 M 中所有停机状态的转换。如果形式化地定义, M' 是根据 M 的成分来定义的:

$$\begin{aligned} Q' &= (Q \cup \{q'_0\}), \Sigma' = \Sigma, \Gamma' = \Gamma, F' = F \\ \delta'(q_i, x) &= \delta(q_i, x), \text{ 如果 } \delta(q_i, x) \text{ 已经定义} \\ \delta'(q'_0, x) &= \delta(q_0, x), \text{ 对于所有的 } x \in \Gamma \\ \delta'(q_i, x) &= [q'_0, x, R], \text{ 如果 } \delta(q_i, x) \text{ 没有被定义} \end{aligned}$$

M' 的初始状态为 q'_0 。如果在 M 上运行输入 w 停机, 那么相应地, M' 就可以再多执行一步计算从而进入 q'_0 。如果 M 上运行输入 w 不停机, 那么在 M' 上就不会执行进入 q'_0 的转换, 从而不会回到初始状态。这样, 我们就把在 M 上运行输入 w 是否停机的问题转换为 M' 是否会重新进入初始状态的问题。因此, 我们便可以得出初始状态重入问题也是不可判定的。

例 12.3.1 我们使用反证法证明了判定对图灵机输入任意的字符串是否会停机是不可判定的。假设有图灵机 A 可以解决这样一个问题: 对于输入的字符串 $v \in \{0, 1, *\}$, 如果输入满足某个图灵机 M

369

的编码, 即 $v = R(M)$; 而且 M 对所有的输入都停机; 则图灵机 A 接收输入。如果输入不能代表一个图灵机, 或者输入所代表的图灵机对于某些输入不能停机, 那么 A 就拒绝接收。

图灵机 A 的计算步骤如下图所示:



下面, 我们将利用归约技术来从图灵机 A 得到图灵机停机问题的解决方法。这样就意味着“对所有字符串停机”问题是不可判定的。

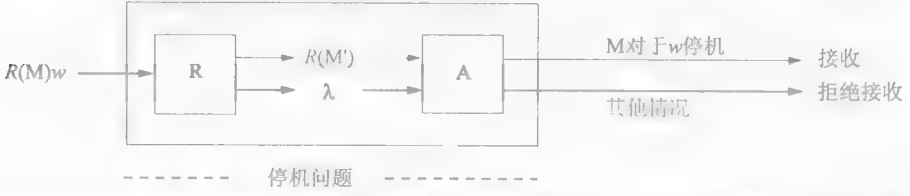
停机问题对应的语言包含了 $R(M)w$ 形式的字符串, 这里 M 代表了某个图灵机, 而且在 M 上运行 w 能够停机。在这里, 我们通过构造图灵机 R 来完成归约的步骤。归约的第一步是判定输入的字符串是否为图灵机的表示及输入字符串的结合。如果输入不具有这样的格式, 那么 R 擦除输入, 并保留空白带。

当输入的字符串为 $R(M)w$ 时, R 的计算构造了这样一个图灵机 M' 的编码, M' 对于输入字符串 y :

- 1. 从带上擦除 y ,
- 2. 将 w 写入到带上,
- 3. 在 M 运行 w 。

$R(M')$ 是通过在 $R(M)$ 基础上增加两组转换规则而得的: 一组转换擦除原本在带上的输入, 另外一组转换在输入的位置写入 w 。 M' 是完全忽略它的输入的。 M' 的计算停机当且仅当在 M 上运行 w 停机。

将图灵机 R 和 A 结合在一起的图灵机如下图所示:



这样就得到了图灵机停机问题的解决方法。如果输入不具有 $R(M)w$ 的格式, 那么利用 R 来产生空字符串, 并由 A 来拒绝。否则的话, 由 R 来产生 $R(M')$ 。追踪图灵机的操作序列, 我们可以知道输入被接收当且仅当它是图灵机 M 和输入字符串 w 的表示, 而且在运行 w 的时候, M 会停机。

因为图灵机停机问题是不可判定的, 而且在 R 上执行归约是可行的, 所以我们可以总结得出没有图灵机能够解决“对所有字符串停机”的问题。 □

370

在 10.1 节中, 我们已经在图灵机和非限定文法之间建立起来了对应关系。这种关系可以用来将非限定的结果从自动机的领域中转换到文法的领域中。通过考察非限定文法 G 是否能够产生字符串 w , 我们便可以借助于归约技术来给出问题的不可判定性。

归 约	输 入	条 件
停机问题	图灵机 M , 字符串 w	M 运行 w 停机
到	↓	当且仅当
推导性问题	非限定文法 G , 字符串 w	G 中存在推导 $S \Rightarrow w$

设 M 为图灵机, 而 w 为输入字符串。归约的第一步是对 M 进行修改, 从而得到另外一个图灵机 M' , M' 能够接收能使 M 停机的每个字符串。为了做到这一点, 我们可以将 M 中所有的状态都设为 M' 的接收状态。在 M' 中, 停机和接收是同样的含义。

利用定理 10.1.3, 我们可以构造文法 G_M , 使它满足 $L(G_M) = L(M')$ 。判定是否 $w \in L(G_M)$ 的算法同样可以判定 M 和 M' 的计算是否停机。因此, 这样的算法是不可能存在的。

12.4 莱斯定理

在前面的章节里面, 我们已经证明了对于任意图灵机的某些特定计算问题, 是不可能构造算法来进行回答的。第一个例子就是图灵机停机问题, 即“图灵机 M 上运行 w 是否会停机?”。通过问题归约技术, 我们可以知道同样没有算法能够回答“从空白带开始, 图灵机 M 是否会停机?”。在这些判定问题中, 输入包含了一个图灵机, 而我们所关心的是图灵机计算的结果。

现在 we 不再考虑图灵机对于特定输入字符串的计算, 而是集中考察图灵机所接收的语言是否满足一个预先确定的属性。例如, 我们可能比较关心如下的这类算法是否存在: 给定一个图灵机 M 作为输入, 能否产生如下问题的解:

- i) $\lambda \in L(M)$?
- ii) $L(M) = \emptyset$?
- iii) $L(M)$ 是正则语言?
- iv) $L(M) = \Sigma^*$?

通过使用在 $\{0, 1\}$ 上的字符串对图灵机进行编码, 我们可以将前面提到的问题转换为语言的成员资格问题。利用编码技术, 能够定义在 $\{0, 1\}$ 上语言的一组图灵机和判断图灵机 M 接收的语言是否满足一个性质的问题, 便可以被视为特定语言 $R(M)$ 的成员问题。例如, $L(M) = \emptyset$ 这个问题可以表示为语言成员问题 $R(M) \in L_\emptyset$ 。使用这种方法, 前面的问题可以转换为:

- i) $L_\lambda = \{R(M) \mid \lambda \in L(M)\}$,
- ii) $L_\emptyset = \{R(M) \mid L(M) = \emptyset\}$,
- iii) $L_{reg} = \{R(M) \mid L(M) \text{ 是正则的}\}$,
- iv) $L_{\Sigma^*} = \{R(M) \mid L(M) = \Sigma^*\}$ 。

例 12.3.1 表明了 L_{Σ^*} 的成员问题是不可判定的。即, 没有算法能够判定图灵机是否对所有的输入字符串都能停机。

例 12.3.1 中使用的归约策略可以被泛化, 进而可以证明很多图灵机表示的语言都不是递归的、递归可枚举语言的一个性质。描述了递归可枚举语言可能满足的条件。例如, L 可能是“语言包含了空串”, “语言是空集”, “语言是正则的”或者“语言包含了所有的字符串”。语言的性质 P 被定义为 $L_P = \{R(M) \mid L(M) \text{ 满足 } P\}$ 。因此, L_\emptyset 描述了所有具有“语言是空集”性质的语言的图灵机表示, 这些图灵机不接收任何输入字符串。

如果没有递归可枚举的语言满足性质 P , 或者所有的递归可枚举语言均满足性质 P , 那么我们将递归可枚举语言的性质 P 称为无价值的 (trivial)。对于每个无价值的性质, L_P 要么是空集要么包含了所有图灵机的表示。这些无价值性质的语言成员资格都是可判定的。莱斯定理 (Rice's Theorem) 证明了对部分递归可枚举语言满足的性质, 它的成员资格却是不可判定的。

定理 12.4.1 (莱斯定理) 如果 L_P 是递归可枚举语言的非无价值性质, 那么 L_P 不是递归的。

证明: 设 P 是递归可枚举语言的非无价值性质。我们将要证明 $L_P = \{R(M) \mid L(M) \text{ 满足 } P\}$ 不是递归的。

因为 L_P 是非无价值的, 那么至少有一个语言 $L \in L_P$ 。而且, 根据空语言不满足 P 的假设, 我们可以知道 L 不为空。这里使用 M_L 来表示接收 L 的图灵机。

我们通过将停机问题归约到 L_P 来证明 L_P 不是递归的。如同在例 12.3.1 中一样, 用于预处理的图灵机 R 被用来完成从输入 $R(M)$ 到图灵机 M' 的转换。那么当输入 y 的时候, M' 的动作为

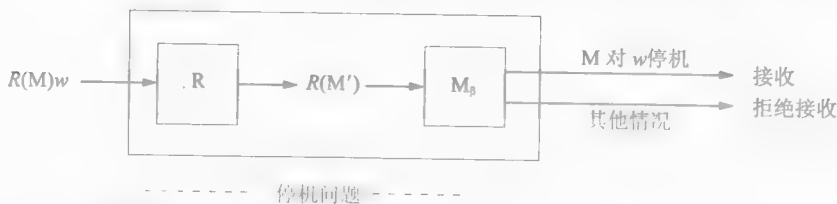
1. 将 w 写入到 y 的右边, 产生 $ByBwB$;
2. 在 M 上运行 w ;
3. 如果 M 运行 w 停机, 那么在 M_L 上运行 y 。

[372] 在这里图灵机 M 和字符串 w 扮演的是看门人的角色。只有当在 M 上运行 w 停机时才允许使用 M_1 来处理 y 。

如果 M 在运行 w 时会停机, 那么允许使用 M_1 来处理 y 。在这种情况下, M' 对于输入 y 的计算动作和 M_1 处理 y 的动作是一样的。因此, $L(M') = L(M_1) = L$ 而且 $L(M')$ 满足 P 。如果 M 对于 w 的计算并不停机, 那么 M' 对于任何输入均不停机。因此, M' 不接收任何输入, 即 $L(M') = \emptyset$, 这样便不满足 P 。

当 M 对于输入 w 不停机时, M' 接收 \emptyset 。当 M 对于输入 w 停机时, M' 接收 L 。既然 L 满足 P , 而 \emptyset 不满足 P , 那么 $L(M')$ 满足 P 当且仅当 M 对于输入 w 停机。

现在假设 L 是递归的。那么必然存在图灵机 M 可以判定 L 的成员资格。将 R 和 M 结合在一起, 我们就可以构造出停机问题的解决方法。



因此, 性质 P 是不可判定的。

我们最初假设空集是不满足 P 的。如果 $\emptyset \in L$, 那么我们便可以用前面的证明来表明 L 不是递归的。这和练习 8.26 中 L 必须是非递归的结论是一致的。■

借助于莱斯定理, 我们可以很容易地利用图灵机接收语言的性质来证明问题的不可判定性。我们将在下面的例子中来展示这一点。

例 12.4.1 “判定一个被图灵机接收的语言是否为上下文无关的”是不可判定的。根据莱斯定理, 我们需要做的工作就是证明“上下文无关”这个性质不是递归可枚举语言的非无价值性质。我们可以通过找到两个递归可枚举的语言, 一个是上下文无关的、另一个不是来完成证明。语言 \emptyset 和 $\{a^i b^j c^k \mid i \geq 0\}$ 都是递归可枚举的。前者是上下文无关的, 而后者却不是。□

12.5 不可解决的词问题

[373] 半图厄 (semi-thue) 系统是根据它的开发人员、挪威数学家阿力克斯·图厄 (Alex Thue) 来命名的。这个系统包含了一个字母表 Σ 和一组规则组成的集合 P 。规则包括了 $u \rightarrow v$, 这里 $u \in \Sigma^+$, $v \in \Sigma^+$ 。这个系统并没有区分变量和终结符, 也没有指定的开始符号。半图厄系统上的词问题是判定对于任意的半图厄系统 $S = (\Sigma, P)$ 和字符串 $u, v \in \Sigma^+$, 是否可以在 S 中从 u 推导出 v 。我们将要证明可以将停机问题归约到词问题。归约是通过在图灵机计算和半图厄系统中的推导之间建立联系来实现的。

设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是确定型图灵机。使用定理 10.1.3 中对于构造过程的修改, 我们可以构造一个模拟 M 计算过程的半图厄系统 $S_M = (\Sigma_M, P_M)$ 。 S_M 的字母表是集合 $Q \cup \Gamma \cup \{[,], q_i, q_R, q_L\}$ 。 S_M 中的规则结合 P_M 定义为:

1. $q_i xy \rightarrow zq_j y$, 当 $\delta(q_i, x) = [q_j, z, R]$ 而且 $y \in \Gamma^+$
2. $q_i x] \rightarrow zq_j B$, 当 $\delta(q_i, x) = [q_j, z, R]$
3. $yq_i x \rightarrow q_j yz$, 当 $\delta(q_i, x) = [q_j, z, L]$ 而且 $y \in \Gamma^+$
4. $q_i x \rightarrow q_R$, 如果 $\delta(q_i, x)$ 有定义
5. $q_R x \rightarrow q_R$, 对于 $x \in \Gamma$
6. $q_R] \rightarrow q_L$
7. $xq_L \rightarrow q_L$, 对于 $x \in \Gamma$
8. $[q_L \rightarrow [q_i$

在定理 10.1.3 中用来生成 $[q_0 B w]$ 的规则在这里被省略了, 这是因为半图厄系统关注的是能否从字符串 u 推导出另一个字符串 v , 而不涉及特定的初始配置。擦除规则 (从 5 到 8) 经过修改, 当

M 对于输入 w 停机的时候, 它们可以产生字符串 $[q_f]$ 。

在 S_M 上模拟 M 的计算需要操作形如 uqv 的字符串, 这里 $u, v \in \Gamma^*, q \in Q \cup \{q_f, q_R, q_L\}$ 。引理 12.5.1 列出了用于模拟 M 计算的 S_M 中推导步骤的一些重要属性。

引理 12.5.1 假设 M 是确定型图灵机, S_M 是从 M 构造的半图厄系统 $w = [uqv]$, 这里 $u, v \in \Gamma^*$, 并且 $q \in Q \cup \{q_f, q_R, q_L\}$ 。

i) 至多有一个字符串 z 满足 $w \xrightarrow{S_M} z$ 。

ii) 如果存在这样的 z , 那么 z 也有 $[u'q'v']$ 的形式, 这里 $u', v' \in \Gamma^*$, 并且 $q' \in Q \cup \{q_f, q_R, q_L\}$ 。

证明: 使用规则可以将 $Q \cup \{q_f, q_R, q_L\}$ 中元素的一个实例替换为另一个。 M 的确定性保证了当 $q \in Q$ 时, 在 P_M 中至多有一个规则可以适用于 $[uqv]$ 。如果 $q = q_R$, 那么仅有一个规则可以适用于 $[uq_Rv]$ 。规则是由字符串 v 中的第一个符号确定。类似地, 也只有一条规则适用于 $[uq_L]$ 。最后, 对于包含 q_f 的字符串, 在 P_M 中没有适用的规则。条件 ii) 则可以直接从 P_M 中的规则得到。 ■

通过在 M 上输入 w 并停机的计算可以得到一个推导

$$[q_0 B w B] \xrightarrow{S_M^*} [u q_R v]$$

擦除规则将这个字符串转化为 q_f 。这些性质合在一起就产生了定理 12.5.2。

定理 12.5.2 确定型图灵机 M 对于输入 w 停机当且仅当 $[q_0 B w B] \xrightarrow{S_M^*} [q_f]$ 。

图灵机计算与相应的半图厄系统中推导之间的关系将利用下面的例子来解释。

例 12.5.1 图灵机



接收的语言是 $0^*1(0 \cup 1)^*$ 。相应的半图厄系统的规则是

$$\begin{array}{lll} q_0 B B \rightarrow B q_1 B & q_1 0 B \rightarrow 0 q_1 B & q_1 1 B \rightarrow 1 q_2 B \\ q_0 B 0 \rightarrow B q_1 0 & q_1 0 0 \rightarrow 0 q_1 0 & q_1 1 0 \rightarrow 1 q_2 0 \\ q_0 B 1 \rightarrow B q_1 1 & q_1 0 1 \rightarrow 0 q_1 1 & q_1 1 1 \rightarrow 1 q_2 1 \\ q_0 B] \rightarrow B q_1 B] & q_1 0] \rightarrow 0 q_1 B] & q_1 1] \rightarrow 1 q_2 B] \\ \\ q_0 0 \rightarrow q_R & q_R B \rightarrow q_R & B q_L \rightarrow q_L \\ q_0 1 \rightarrow q_R & q_R 0 \rightarrow q_R & 0 q_L \rightarrow q_L \\ q_1 B \rightarrow q_R & q_R 1 \rightarrow q_R & 1 q_L \rightarrow q_L \\ q_2 B \rightarrow q_R & q_R] \rightarrow q_L] & [q_L \rightarrow [q_f \\ q_2 0 \rightarrow q_R & & \\ q_2 1 \rightarrow q_R & & \end{array}$$

[375]

M 接收 011 的计算对应于半图厄系统 S_M 中从 $[q_0 B 011 B]$ 到 $[q_f]$ 的推导。

$$\begin{array}{ll} q_0 B 011 B & [q_0 B 011 B] \\ \vdash B q_1 011 B & \Rightarrow [B q_1 011 B] \\ \vdash B 0 q_1 11 B & \Rightarrow [B 0 q_1 11 B] \\ \vdash B 0 1 q_2 1 B & \Rightarrow [B 0 1 q_2 1 B] \\ & \Rightarrow [B 0 1 q_R B] \\ & \Rightarrow [B 0 1 q_R] \\ & \Rightarrow [B 0 1 q_L] \end{array}$$

$$\Rightarrow [B0q_L]$$

$$\Rightarrow [Bq_L]$$

$$\Rightarrow [q_L]$$

$$\Rightarrow [q_f]$$

□

利用半图厄系统中的推导来模拟图灵机中计算的能力, 为我们证明半图厄系统中词问题的不可判定性提供了基础。

定理 12.5.3 半图厄系统中的词问题是不可判定的。

证明: 前面的定理已经给出了从停机问题到词问题的归约过程。对于一个图灵机 M 和它相应的半图厄系统 S_M , 图灵机 M 针对 w 的计算停机等同于可以在 S_M 中从 q_0BwB 推导出 $[q_f]$ 。解决词问题的算法同样可以用来解决图灵机停机问题。 ■

根据定理 12.5.3, 我们知道对于任意的半图厄系统 $S = (\Sigma, P)$ 和 Σ^* 中的一对字符串, 没有算法能够解决词问题。利用我们在定理 12.5.2 中建立的图灵机 M 计算与 S_M 系统中推导之间的关系, 同样可以证明在特定的半图厄系统中, 词问题也是不可判定的。

定理 12.5.4 设 M 为接收非递归语言的确定型图灵机。那么在半图厄系统 S_M 中, 词问题是不可判定的。

证明: 因为 M 识别的是非递归语言, 所以对于 M 来说, 停机问题是不可判定的 (练习 3)。 M 的计算与 S_M 的推导之间的对应关系表明了这个系统中词问题是不可判定的。 ■

376

12.6 波斯特对应问题

在前面的章节中引入的不可判定问题与图灵机的性质或者与模拟图灵机的数学系统有关联。波斯特对应问题是一个复合的问题; 它可以被描述为操作多米诺骨牌的游戏。

aba
bhaba

一个多米诺骨牌包含了从固定字母表得出的两个非空字符串, 一个在多米诺骨牌的上半部分, 另一个在下半部分。

波斯特对应系统可以被视为定义了一个有限类型的多米诺骨牌的集合。

游戏从放在桌子上的一个多米诺骨牌开始。接着在它的右边再安放另外一个多米诺骨牌。重复这个过程, 从而产生一系列彼此相邻近的多米诺骨牌。这里我们假设每种多米诺骨牌的数目是无穷的。

将一系列多米诺骨牌的上半部分连接在一起就可以得到字符串。我们把它称为顶部字符串。类似地可以定义底部字符串。游戏的目的是要找到一个序列, 从而能够产生同样的顶部字符串和底部字符串。考虑波斯特对应系统中有下列的多米诺骨牌

a	c	ba	acb
ac	ba	a	b

下面的序列能够产生 *acbaaacb* 这样等价的顶部和底部字符串

a	c	ba	a	acb
ac	ba	a	ac	b

严格地说, 波斯特对应系统 (post correspondence system) 包含了字母表 Σ 和有限的有序对 $[u_i, v_i]$ 集合, 其中 $i = 1, 2, 3, \dots, n, u_i, v_i \in \Sigma^*$ 。波斯特对应系统的解决方法是一个序列 i_1, i_2, \dots, i_k , 这个序列满足

$$u_{i_1}u_{i_2}\cdots u_{i_k} = v_{i_1}v_{i_2}\cdots v_{i_k}$$

判定波斯特对应系统是否存在这样的解决方法的问题被称为波斯特对应问题。

377

例 12.6.1 拥有字母表上 $\{a, b\}$ 和有序对 $[aaa, aa], [baa, abaaa]$ 的波斯特对应系统, 具有如下的解决方法。

aaa	baa	aaa
aa	abaaa	aa

□

例 12.6.2 对于具有字母表 $\{a, b\}$ 和有序对 $[ab, aba]$, $[bba, aa]$, $[aba, bab]$ 的波斯特对应系统来说, 如果解决方法存在的话, 必然是以下面的多米诺开始

ab
aba

因为这个是惟一的顶部和底部前缀一致的多米诺骨牌。顶部的字符串接下来必须以 a 开始。有两种可能:

ab	ab
aba	aba

(a)

ab	aba
aba	bab

(b)

在图 (a) 中的第四个字符不再匹配。那么构造解决方法的惟一途径就是对图 (b) 进行扩展。同样的原因, 我们看到在顶部字符串中下一个要出现的元素必须是 b 。因此, 就会产生

ab	aba	bba
aba	bab	aa

但是, 因为顶部和底部的第七个字符不相同, 所以不可能构造一个解决方法。我们已经表明了没有办法能够构造出顶部和底部相同的字符串。因此, 这个波斯特对应系统并不存在解决方法。□

通过在半图厄系统中利用推导来模拟多米诺骨牌的序列, 我们可以证明波斯特对应问题是不可判定的。根据定理 12.5.4, 我们已经知道半图厄系统 $S = (\Sigma, P)$ 的词问题是不可判定的。即, 没有算法能够判定是否可以通过使用规则 P , 从字符串 u 推导出字符串 v 。相应的归约如下。

归 约	输 入	条 件
$S = (\Sigma, P)$ 中的可推导性	字符串 u, v	从 u 可以推导出 v
到	↓	当且仅当
波斯特对应系统	多米诺骨牌集合 $C_{u,v}$	波斯特对应系统 $C_{u,v}$ 存在解决方法

归约过程包含根据规则 P 和字符串 u 和 v 来在半图厄系统中进行推导, 从而产生多米诺骨牌。

定理 12.6.1 没有算法能够判定任意的一个波斯特对应系统是否存在解决方法。

证明: 我们用 $S = (\Sigma, P)$ 来表示定义在 $\{0, 1\}$ 上, 而且词问题是不可判定的半图厄系统。对于字符串 $u, v \in \Sigma^*$, 我们可以构造相应的波斯特对应系统 $C_{u,v}$, $C_{u,v}$ 有解决方法当且仅当 $u \xRightarrow{P} v$ 。因为后面的这个问题是不可判定的, 所以就没有通用的算法能够解决波斯特对应问题。

我们首先在 S 的产生式集合中加入规则 $0 \rightarrow 0$ 和 $1 \rightarrow 1$ 。在加入规则后的系统中, 大部分推导都和 S 相同, 只是增加了不对字符串进行转换的规则。然而, 使用这种规则可以保证当 $u \xRightarrow{P} v$ 时, u 可以经过等长度的推导得出 v 。为了避免符号混用, 这里我们继续使用符号 S 来表示加入规则后的系统。

假设 u 和 v 是 $\{0, 1\}^*$ 上的字符串。波斯特对应系统 $C_{u,v}$ 由 u, v 和 S 构造得到。 $C_{u,v}$ 的字母表包含了 $0, \bar{0}, 1, \bar{1}, [,], *, \bar{*}$ 。全部由包含上标的字符组成的字符串 w , 可以被表示为 \bar{w} 。

S (包含了 $0 \rightarrow 0, 1 \rightarrow 1$) 中的每个产生式 $x_i \rightarrow y_i, i = 1, 2, 3, \dots, n$, 定义了两个多米诺骨牌

\bar{y}_i	y_i
x_i	x_i

由多米诺骨牌组成的系统是

\bar{u}^*	$*$	$*$	$*$
$[$	$*$	$*$	$[\bar{*}v]$

多米诺骨牌

0	$\bar{0}$	1	$\bar{1}$
$\bar{0}$	0	$\bar{1}$	1

可以被组合起来, 从而获得由

w	\bar{w}
\bar{w}	w

构成的多米诺骨牌序列, 其中任意的 $w \in \{0, 1\}^*$. 当构造波斯特对应系统 $C_{u,v}$ 的解决方法的时候, 我们可以自由地使用这些组合的多米诺骨牌。

首先我们证明当 $u \xrightarrow{*} v$ 时, $C_{u,v}$ 存在解决方法。设

$$u \Rightarrow u_0 \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

是一个等长的推导。规则 $0 \rightarrow 0$ 和 $1 \rightarrow 1$ 确保了当可以从 u 推导出 v 的时候, 存在一个等长的推导。推导的第 i 步可以被写成

$$u_{i-1} = p_{i-1} x_{j_{i-1}} q_{i-1} \Rightarrow p_{i-1} y_{j_{i-1}} q_{i-1} = u_i$$

这里使用规则 $x_{j_{i-1}} \rightarrow y_{j_{i-1}}$ 来从 u_{i-1} 推导出 u_i 。字符串

$$[u_0 * \bar{u}_1 * \bar{u}_2 * \bar{u}_3 * \cdots * \bar{u}_{k-1} * \bar{u}_k]$$

就是 $C_{u,v}$ 的解决方法。可以利用下面的方法来构造解决方法。

1. 开始的时候

u^*

2. 为了获得匹配, 能产生底部 $u = u_0$ 的多米诺骨牌, 需要进行如下的组合,

$[u^*$	\bar{p}_0	\bar{y}_{j_0}	\bar{q}_0	*
[p_0	x_{j_0}	q_0	*

[380] 由 p_0 和 q_0 构成的多米诺骨牌都是复合多米诺骨牌。中间的多米诺骨牌由规则 $x_{j_0} \rightarrow y_{j_0}$ 来产生。

3. 因为 $p_0 y_{j_0} q_0 = u_1$, 那么上面的字符串必须写上 $[u_0 * \bar{u}_1]$, 下面的字符串要写上 $[u_0]$ 。重复上面的策略, 我们必须在底部拼写出 \bar{u}_1

$[u^*$	p_0	\bar{y}_{j_0}	\bar{q}_0	*	p_1	y_{j_1}	q_1	*
[p_0	x_{j_0}	q_0	*	p_1	\bar{x}_{j_1}	\bar{q}_1	\bar{u}_1

这将在顶部生成 $[u_0 * \bar{u}_1 * \bar{u}_2 * \cdots]$ 。

4. 重复推导过程中的步骤 2、3、 \cdots 、 $k-1$ 就可以产生

$[u^*$	p_0	\bar{y}_{j_0}	\bar{q}_0	*	p_1	y_{j_1}	q_1	*	\cdots	p_{k-1}	$y_{j_{k-1}}$	q_{k-1}
[p_0	x_{j_0}	q_0	*	p_1	\bar{x}_{j_1}	\bar{q}_1	\bar{u}_1	\cdots	\bar{p}_{k-1}	$\bar{x}_{j_{k-1}}$	\bar{q}_{k-1}

5. 通过下面的多米诺骨牌

]
$u^* v$

来完成这个序列, 从而能够在底部和顶部产生 $[u_0 * \bar{u}_1 * \bar{u}_2 * \cdots * \bar{u}_{k-1} * \bar{u}_k]$, 从而解决了波斯特对应问题。

我们将证明可以通过波斯特对应系统 $C_{u,v}$ 的解决方法来得到从 $u \Rightarrow v$ 的推导。 $C_{u,v}$ 的解决方法必须以下面的多米诺骨牌开始

$u *$

因为这是惟一以相同符号开始的多米诺骨牌。同样的原因,解决方法也必须以下面的多米诺骨牌结束

$]$
$* v$

因此,解决方法中的字符串应该是 $[u * w * v]$ 。如果 w 包含了 $]$,那么解决方法必须是 $[u * x * v] y * v]$ 。因为 $]$ 仅出现在一个多米诺骨牌中,而且还是那个多米诺骨牌中上部和下部最右边的符号。因此, $[u * w * v]$ 同样也是 $C_{u,v}$ 的解决方法。

在前面的观察中,设 $u * \dots * v$ 是波斯特对应系统 C_u 的解决方法,在这里, $]$ 只作为最右边的符号出现。多米诺骨牌所提供的信息决定了整个解决方法的结构,解决方法以下面的多米诺骨牌开始。 [381]

$[u *$
$[$

因为上面已经产生了 u ,所以多米诺骨牌的序列能够拼写出 u 。设 $u = x_1 x_2 \dots x_k$ 是解决方法中能够在底部拼写出 u 的序列。那么解决方法应该是:

$u *$	\bar{y}_1	y_{i_2}	y_{i_3}	\dots	\bar{y}_{i_k}	$*$
	x_{i_1}	x_{i_2}	x_{i_3}	\dots	x_{i_k}	$*$

因为每个多米诺骨牌都代表了推导 $x_i \Rightarrow y_i$,所以我们将这些结合在一起就能够得到 $u \Rightarrow u_1$,这里 $u_1 = y_{i_1} y_{i_2} \dots y_{i_k}$ 。在构成解决方法的多米诺骨牌中,它们所组成的顶部的字符串具有 $[u * \bar{u}_1 * \dots]$ 形式的前缀,而底部的字符串则有 $[u * \dots]$ 形式的前缀。重复这个过程,我们可以看到解决方法定义了下面的字符串序列。

$$\begin{aligned} & [u * \bar{u}_1 * \bar{u}_2 * \dots * \bar{u}_k * v] \\ & [u * \bar{u}_1 * \bar{u}_2 * \bar{u}_3 * \dots * \bar{u}_k * v] \\ & [u * \bar{u}_1 * \bar{u}_2 * \bar{u}_3 * \bar{u}_4 * \dots * \bar{u}_k * v] \\ & \vdots \\ & [u * \bar{u}_1 * \bar{u}_2 * \bar{u}_3 * \bar{u}_4 * \dots * \bar{u}_{k-1} * \bar{u}_k * v] \end{aligned}$$

其中 $u_i \Rightarrow u_{i+1}$, $u_0 = u$ 而且 $u_k = v$ 。将这些组合在一起可以得到推导 $u \Rightarrow v$ 。

前面的两个证明构成了从半图厄系统中的词问题到波斯特对应问题的归约。这样就证明了波斯特对应问题是不可判定的。 ■

12.7 上下文无关文法中的不可判定问题

上下文无关文法为定义编程语言的语法提供了重要的工具。波斯特对应问题的不可判定性可以被用来确定一些与上下文无关文法相关的重要问题的不可判定性。为了在波斯特对应系统和上下文无关文法之间建立起来联系,我们使用波斯特对应问题中的多米诺骨牌用来定义两个上下文无关文法中的规则。

设 $C = (\Sigma_C, \{[u_1, v_1], [u_2, v_2], \dots, [u_n, v_n]\})$ 是一个波斯特对应系统。我们从 C 的有序对出发可以得到两个上下文无关的文法 G_0 和 G_1 :

[382]

$$\begin{aligned}
G_U: V_U &= \{S_U\} \\
\Sigma_U &= \Sigma_C \cup \{1, 2, \dots, n\} \\
P_U &= \{S_U \rightarrow u_i S_U i, S_U \rightarrow u_i i \mid i = 1, 2, \dots, n\} \\
G_L: V_L &= \{S_L\} \\
\Sigma_L &= \Sigma_C \cup \{1, 2, \dots, n\} \\
P_L &= \{S_L \rightarrow v_i S_L i, S_L \rightarrow v_i i \mid i = 1, 2, \dots, n\}
\end{aligned}$$

可以将判定波斯特对应系统 C 是否有解决方法的问题归约到相应的文法 G_U 和 G_L 中是否具有可推导性。文法 G_U 生成的是将在多米诺骨牌序列中上半部分出现的字符串。规则中的数字将记录产生字符串的多米诺骨牌的顺序（或者相反的顺序）。类似地， G_L 中产生的字符串则是在多米诺骨牌序列中的下半部分出现。

如果存在序列 $i_1 i_2 \dots i_{k-1} i_k$ 使得

$$u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k}$$

那么波斯特对应系统 C 中就存在解决方法。在这种情况下， G_U 和 G_L 中包含了归约

$$\begin{aligned}
S_U &\xrightarrow{G_U} u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} i_{k-1} \dots i_2 i_1 \\
S_L &\xrightarrow{G_L} v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k} i_{k-1} \dots i_2 i_1
\end{aligned}$$

其中， $u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} i_{k-1} \dots i_2 i_1 = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k} i_{k-1} \dots i_2 i_1$ 。因此， $L(G_U)$ 和 $L(G_L)$ 的交集不为空。

相应的，假设 $w \in L(G_U) \cap L(G_L)$ 。那么 w 包含了字符串 $w' \in \Sigma_C^+$ 以及序列 $i_1 i_{k-1} \dots i_2 i_1$ 。字符串 $w' = u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k}$ 是 C 的解决方法之一。

例 12.7.1 文法 G_U 和 G_L 是从例 12.6.1 中的波斯特对应系统 $\{aaa, aa\}$ 和 $\{baa, abaaaa\}$ 中得来的。

$$\begin{array}{ll}
G_U: S_U \rightarrow aaa S_U 1 & G_L: S_L \rightarrow aa S_L 1 \\
\rightarrow baa S_U 2 & \rightarrow abaaaa S_L 2
\end{array}$$

能够给出波斯特对应问题解法的推导是这样的：

$$\begin{array}{ll}
S_U \Rightarrow aaa S_U 1 & S_L \Rightarrow aa S_L 1 \\
\Rightarrow aaabaa S_U 21 & \Rightarrow aaabaaaa S_L 21 \\
\Rightarrow aaabaaaaaa 121 & \Rightarrow aaabaaaaaa 121.
\end{array}$$

□

波斯特对应问题的解法与文法 G_U 和 G_L 中推导之间的关系可以被用来证明一些由上下文无关文法得到的问题的不可判定性。

定理 12.7.1 没有算法能够判定两个上下文无关文法产生的语言是否是分离的（即交集为空）。

证明：假设这样的算法存在，那么波斯特对应问题便可以通过下面的步骤来解决：

1. 对于任意的波斯特对应系统 C ，根据 C 中的有序对来构造文法 G_U 和 G_L 。
2. 利用算法来判定 $L(G_U) \cap L(G_L)$ 是否为空。
3. C 存在解决方法当且仅当 $L(G_U) \cap L(G_L)$ 不为空。

第一步将波斯特对应问题归约到判定波斯特系统产生了两个上下文无关文法的语言是否分离。因为波斯特对应系统已经被证明是不可判定的，所以我们可以总结得出上下文无关文法产生的语言是否分离也是不可判定的。■

定理 12.7.2 没有算法能够判定任意的上下文无关文法是否是二义的。

证明：如果在上下文无关文法中，一个字符串可以用两种不同的推导方式得到，那么该上下文无关文法是二义的。如前面一样，我们从波斯特对应系统 C 开始，并构造文法 G_U 和 G_L 。将这些文法结合在一起就可以得到文法

$$\begin{aligned}
G: L &= \{S, S_U, S_L\} \\
\Sigma &= \Sigma_U \\
P &= P_U \cup P_L \cup \{S \rightarrow S_U, S \rightarrow S_L\}
\end{aligned}$$

这个文法从 S 开始, 可以生成 $L(G_L) \cup L(G_U)$ 。

G 中所有的推导都在最左边, 每个语句形式都至多包含一个变量。 G 中的推导包含了使用 S 中的规则, 并使用 G_U 或 G_L 中的推导。文法 G_U 和 G_L 是无二义的——不同的推导生成了不同的后缀。这意味着 G 是无二义的当且仅当 $L(G_U) \cap L(G_L)$ 不为空。但是这个条件等同于在初始的波斯特对应系统 C 中存在相应的解决方法。既然波斯特对应问题可以归约到判定一个上下文无关文法是否是二义的, 那么后者肯定是不可判定的。 ■

在第 7.5 节中, 我们看到上下文无关语言家族在完整性上不是闭合的。然而, 对于任意的波斯特对应系统 C , $L(G_U)$ 和 $L(G_L)$ 都是上下文无关的。我们可以使用这个性质来证明下面两个问题的不可判定性: 对于任意的上下文无关文法是否能够产生字母表上所有的字符串, 以及是否两个上下文无关的文法产生了同样的语言。

定理 12.7.3 没有算法能够判定上下文无关文法 $G = \{L, \Sigma, P, S\}$ 的语言是否为 Σ^* 。

证明: 首先, 注意 $L = \Sigma^*$ 等同于 $L = \emptyset$ 。我们将证明没有算法能够判定对于任意的上下文无关文法 $L(G)$ 是否为空。

设 C 是波斯特对应系统, 相对应的文法是 G_U 和 G_L 。能够产生 $L(G_U) \cup L(G_L)$ 的文法 G' 可以直接从产生 $L(G_U)$ 和 $L(G_L)$ 的上下文无关文法得到。根据德摩根定律, $L(G') = L(G_U) \cup L(G_L)$ 。

判定任意上下文无关文法的 $L(G) = \emptyset$ 的算法可以用来解决波斯特对应问题:

1. 对于波斯特对应系统 C , 构造文法 G_U 和 G_L 。
2. 构造能够产生 $L(G_U)$ 和 $L(G_L)$ 的文法。
3. 从能够产生 $L(G_U)$ 和 $L(G_L)$ 的文法中构造 G' 。
4. 使用判定算法来判定 $L(G') = \emptyset$ 是否成立。
5. $L(G) = \emptyset$ 当且仅当 $L(G_U)$ 和 $L(G_L)$ 是分离的, 当且仅当 C 有解决方法。

因此没有算法能够判定 $L(G) = \emptyset$ 是否成立, 相应地, 也不能判定 $L(G) = \Sigma^*$ 是否成立。 ■

定理 12.7.4 没有算法能够判定两个上下文无关文法产生的语言是否是等价的。

证明: 设 C 是波斯特对应系统, 相对应的文法是 G_U 和 G_L 。如同在定理 12.7.3 中的证明, 可以构造上下文无关的文法 G_U , 来产生 $L(G_U) \cup L(G_L) = L(G_U) \cap L(G_L)$ 。第二个上下文无关文法 G_L 产生了 Σ_U 上所有的字符串。

$L(G_U)$ 包含了 Σ_U 中所有不是波斯特对应系统解决方法的字符串。因此, $L(G_U) = L(G_L)$ 当且仅当 C 中不包含问题的解决方法。相应地, 判定两个文法产生的语言是否相同的算法也可以被用来判定波斯特对应问题是否有解。 ■

12.8 练习

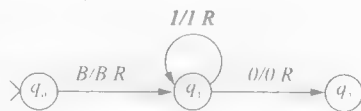
1. 证明通用图灵机上的停机问题是不可判定的。即, 没有图灵机能够判定对于通用图灵机 U 和任意的输入, 计算会不会停机。
2. 解释例 11.5.2 中 n 次转换停机问题和图灵机停机问题区别, 为什么前者是可判定的, 而后者是不可判定的。
3. 设 M 是接受非递归语言的确定型图灵机。证明 M 的停机问题是不可判定的。即, 没有图灵机能够判定, 对于输入 w , 图灵机对于 w 是否停机。

在练习 4 到练习 8 中, 使用归约来建立判定问题的不可判定性。

4. 证明没有算法能够判定对于任意的图灵机, 当输入 101 时, 计算是否会停机。
5. 证明没有算法能够判定对于任意的图灵机, 是否会对至少一个输入字符串停机。
6. 证明没有算法能够判定对于图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, 状态 $q_i \in Q$, 字符串 $w \in \Sigma^*$, 针对 w 的计算是否会进入状态 q_i 。
7. 证明没有算法能够判定对于任意图灵机, 在最终的转换过程中是否会在带上打印 1。

[386]

8. 证明没有算法能够判定对于任意的图灵机, 当从空白带开始运行时, 是否会在三次连续的转化过程中打印 1。
9. 为什么我们不能用下面的方式来证明空白带停机问题是不可判定的: 空白带停机问题是停机问题的子问题, 而停机问题是不可判定的, 因此它也是不可判定的。
10. 证明判断在 $\Sigma = \{1\}$ 上的字符串长度是否为偶数可以归约到空白带停机问题。为什么无法从这里得出判定字符串长度是否为偶数是不可判定的。
11. 给出一个任意递归可枚举语言都不满足的性质。
12. 使用莱斯定理来证明下面递归可枚举语言的性质是不可判定的。为了建立这种不可判定性, 所要做的就是证明这些性质不是无价值的。
 - a) L 包含了特定的字符串 w_0 。
 - b) L 是无穷的。
 - c) L 是正则的。
 - d) L 是 $\{0, 1\}^*$ 。
13. $L = \{R(M) \mid M \text{ 运行 } R(M) \text{ 时停机}\}$ 。
 - a) 证明 L 不是递归的。
 - b) 证明 L 是递归可枚举的。
- * 14. $L_{\neq \emptyset} = \{R(M) \mid L(M) \text{ 非空}\}^*$ 。
 - a) 证明 $L_{\neq \emptyset}$ 不是递归的。
 - b) 证明 $L_{\neq \emptyset}$ 是递归可枚举的。
15. 设 M 是图灵机
 - a) 给出半图厄系统 S_M 中能够模拟 M 计算的规则。
 - b) 跟踪当输入 01 时 M 的计算过程, 并给出在 S_M 中相应的推导
16. 找到下列波斯特对应系统中的解法
 - a) $[a, aa], [bb, b], [a, bb]$
 - b) $[a, aaa], [aab, b], [abaa, ab]$
 - c) $[aa, aab], [bb, ba], [abb, b]$
 - d) $[a, ab], [ba, aba], [b, aba], [bba, b]$
17. 证明下列波斯特对应系统不存在解
 - a) $[b, ba], [aa, b], [bab, aa], [ab, ba]$
 - b) $[ab, a], [ba, bab], [b, aa], [ba, ab]$
 - c) $[ab, aba], [baa, aa], [aba, baa]$
 - d) $[ab, bb], [aa, ba], [ab, abb], [bb, bab]$
 - e) $[abb, ab], [aba, ba], [aab, abab]$



[387]

- * 18. 证明对于仅包含一个符号的字母表上的波斯特系统, 其对应问题是可判定的。
19. 假设 P 是定义在 $[b, bbb], [babbb, ba], [bab, aab]$ 和 $[ba, a]$ 上的波斯特对应问题。
 - a) 给出 P 的解决方法。
 - b) 从 P 构造 G_U 和 G_L 。
 - c) 给出 (a) 中解决方法在 G_U 和 G_L 上对应的推导。
20. 利用波斯特对应系统 $[b, bb], [aa, baa]$ 和 $[ab, a]$ 来构造上下文无关文法 G_U 和 G_L 。那么 $L(G_U) \cap L(G_L) = \emptyset$ 是否成立?
- * 21. 设 C 是波斯特对应系统, 构造一个可以产生 $\overline{L(G_U)}$ 的上下文无关文法。
- * 22. 证明没有算法能够判定两个上下文无关文法产生的语言的交集, 是否包含有限个元素。
23. 证明没有算法能够判定上下文无关文法产生的语言是否包含有限个元素。
- * 24. 证明没有算法能够判定两个上下文无关文法 G_1 和 G_2 产生的语言, 是否满足 $L(G_1) \subseteq L(G_2)$ 。

参考文献注释

停机问题的不可判定性由图灵 [1936] 证明。在 12.1 节中给出的证明来自 Minsky [1967]。利用语言的性质来得出问题的不可判定性是由莱斯提出的 [1953] 和 [1956]。Thue 中的字符串转换系统由 Thue [1914] 引入。半图厄系统中词问题的不可判定性由 Post 得出 [1947]。

波斯特对应问题的不可判定性由 Post [1946] 提出。定理 12.6.1 的证明是基于 Floyd [1964] 的技术，来自 Davis 和 Weyuker [1983]。上下文无关语言的不可判定性，包括定理 12.7.1，可以在 Bar-Hillel、Perles 和 Shamir [1961] 中找到。上下文无关语言二义性的不可判定性由 Cantor [1962]、Floyd [1962] 和 Chomsky、Schutzenberger [1963] 提出。固有的二义性问题由 Ginsburg 和 Ullman [1966a] 证明是不可判定的。

第 13 章 μ -递归函数

在第 9 章中,我们从机器的角度介绍了可计算性,通过图灵机的转换可以得到函数的计算值。丘奇-图灵论题证明了每个可以计算的函数都可以通过这种方式来实现。但是究竟什么样的函数才是图灵可计算的呢?在本章中,我们将引入这个问题的答案,并将进一步地支持丘奇-图灵论题。

我们已经从宏观的角度考察了可计算函数。下面,我们将具体考察函数而不是图灵机计算的具体操作。我们将引入两组函数:原始递归函数和 μ -递归函数。原始递归函数是由一组可计算函数通过组合操作和原始递归操作来构造的。而 μ -递归函数则是通过增加无限制的最小化和顺序查询的功能表示构造得来的。

原始递归函数和 μ -递归函数的可计算性是由能够有效计算函数值的方法来展示的,对于有效计算的分析是通过图灵可计算性和 μ -递归性的等价来完成的。这也就回答了开篇提出的问题——能够被图灵机计算的函数都是 μ -递归函数。

13.1 原始递归函数

[389] 原始递归函数,也就是可以直接计算的数论函数家族,是通过下面的基本函数:

- i) 后继函数 $s; s(x) = x + 1$
- ii) 零函数 $z; z(x) = 0$
- iii) 映射函数 $p_i^n; p_i^n(x_1, \dots, x_n) = x_i, 1 \leq i \leq n$

以及将已经在家族中的函数利用操作符构造出的新函数来完成的。基本函数的简单性保证了它们的可计算性。后继函数只需要在自然数上加 1。零函数的计算更是简单,对所有的参数其结果均为 0。映射函数 p_i^n 的返回值就是它的第 i 个参数。

原始递归函数是由基础函数使用两种操作构造而得到的,这两种操作均可以保持它们的可计算性。第一个是组合(在 9.4.2 节定义)。设 f 是由具有 n 个参数的函数 h 和具有 k 个参数的函数 g_1, g_2, \dots, g_n 组合的。如果组合中的每个成分都是可计算的,那么 $f(x_1, \dots, x_k)$ 的值可以从 h 和 $g_1(x_1, \dots, x_k), g_2(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)$ 来得到。 f 的可计算性是由它成分函数的可计算性得来的。产生新函数的第二种操作就是原始递归。

定义 13.1.1 设 g 和 h 分别是具有 n 和 $n+2$ 个变量的数论函数。那么具有 $n+1$ 个变量的函数 f 定义如下:

- i) $f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$
- ii) $f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$

这样我们便称 f 为从 g 和 h 进行了原始递归(primitive recursion)。其中 x_i 被称为原始递归的参数。变量 y 是递归变量。

当 g 和 h 可计算的时候,原始递归的操作提供了计算 $f(x_1, \dots, x_n, y)$ 值的算法。对于一组固定的参数 x_1, \dots, x_n , $f(x_1, \dots, x_n, 0)$ 可以直接从函数 g 中得到:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$f(x_1, \dots, x_n, y+1)$ 的值是由可计算函数 h 使用下列信息得到的:

- i) 参数 x_1, \dots, x_n ,
- ii) y , 递归变量的前一个值, 以及
- iii) $f(x_1, \dots, x_n, y)$, 函数的前一个值。

例如,我们可以从下面的计算中得到 $f(x_1, \dots, x_n, y+1)$

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$\begin{aligned}
 f(x_1, \dots, x_n, 1) &= h(x_1, \dots, x_n, 0, f(x_1, \dots, x_n, 0)) \\
 f(x_1, \dots, x_n, 2) &= h(x_1, \dots, x_n, 1, f(x_1, \dots, x_n, 1)) \\
 &\vdots \\
 f(x_1, \dots, x_n, y+1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)).
 \end{aligned}$$

因为 h 是可计算的, 所以这个迭代的过程可以用来确定对于任意的递归变量 y , $f(x_1, \dots, x_n, y+1)$ 的值。

定义 13.1.2 如果一个函数可以从后继函数、零函数和映射函数经过一系列的组合和原始递归而得到, 那么该函数就可以被称为是原始递归的 (primitive recursive)。

经过一系列的组合以及原始递归得到的完全函数仍然还是完全的。这一点可以从操作的定义中直接得到, 我们把证明过程留作练习。因为基本的原始递归函数都是完全的, 而操作保持了完全性, 所以所有的原始递归函数都是完全的。

我们将组合和原始递归结合在一起可以得到构造函数的强有力工具。接下来的例子证明了对于任意的常数函数, 加、乘和阶乘, 都是原始递归函数。

例 13.1.1 常数函数 $c_i^{(n)}(x_1, \dots, x_n) = i$ 都是原始递归函数。例 9.4.2 定义了如何将常数函数视为后继函数、零函数和映射函数的组合。□

例 13.1.2 我们可以将 add 函数视为 $g(x) = x$ 和 $h(x, y, z) = z + 1$ 的原始递归。那么

$$\begin{aligned}
 add(x, 0) &= g(x) = x \\
 add(x, y+1) &= h(x, y, add(x, y)) = add(x, y) + 1
 \end{aligned}$$

add 函数计算了两个自然数的和。 $add(x, 0)$ 的定义表明了任何数与零的和都是自身。后面的等式定义了 x 和 $y+1$ 的和是将 x 和 y 的和 (add 函数中对于递归变量的前一个结果) 再加一。

前面的定义说明了加函数是原始递归的。根据原始递归的定义, g 和 h 也都是原始递归的, 因为 $g = p_1^{(1)}$ 而且 $h = s \circ p_3^{(3)}$ 。

两个自然数进行加的结果可以从 add 函数的原始递归定义中得出, 我们可以重复地执行条件 $add(x, y+1) = add(x, y) + 1$ 来归纳出递归变量的值。例如:

$$\begin{aligned}
 add(2, 4) &= add(2, 3) + 1 \\
 &= (add(2, 2) + 1) + 1 \\
 &= ((add(2, 1) + 1) + 1) + 1 \\
 &= (((add(2, 0) + 1) + 1) + 1) + 1 \\
 &= (((2 + 1) + 1) + 1) + 1 \\
 &= 6
 \end{aligned}$$

当递归变量成为零的时候, 函数 g 就可以用来初始化表达式的计算。□

例 13.1.3 设 g 和 h 都是原始递归函数, $g = z, h = add \circ (p_1^{(1)}, p_1^{(1)})$ 。乘法可以利用 g 和 h 进行原始递归来定义:

$$\begin{aligned}
 mult(x, 0) &= g(x) = 0 \\
 mult(x, y+1) &= h(x, y, mult(x, y)) = mult(x, y) + x
 \end{aligned}$$

插入的表达式对应了原始递归的定义, $x \cdot (y+1) = x \cdot y + x$, 这也对应了加和乘的分配性。□

我们可以根据 13.1.1 中的定义, 使用原始递归和两个变量的函数 h 来定义一个变量的函数。这样的函数 f 定义如下

- i) $f(0) = n_0$, 这里 $n_0 \in \mathbb{N}$
- ii) $f(y+1) = h(y, f(y))$

例 13.1.4 仅有一个变量的函数

$$fact(y) = \begin{cases} 1 & \text{如果 } y = 0 \\ \prod_{i=1}^y i & \text{否则} \end{cases}$$

是原始递归的。设 $h(x, y) = mult \circ (p_2^{(2)}, s \circ p_1^{(2)}) = y \cdot (x+1)$ 。可以使用从 h 开始的原始递归定义来

[391]

[392]

定义阶乘函数

$$\text{fact}(0) = 1$$

$$\text{fact}(y+1) = h(y, \text{fact}(y)) = \text{fact}(y) \cdot (y+1).$$

注意, 这里的定义采用的递归变量的值是 $y+1$ 。这可以通过 y 的后继函数, 提供给 h 函数的值来得到。

下面列出的 fact 函数的前五个输入值对应的结果, 解释了如何使用原始递归定义来执行计算:

$$\text{fact}(0) = 1$$

$$\text{fact}(1) = \text{fact}(0) \cdot (0+1) = 1$$

$$\text{fact}(2) = \text{fact}(1) \cdot (1+1) = 2$$

$$\text{fact}(3) = \text{fact}(2) \cdot (2+1) = 6$$

$$\text{fact}(4) = \text{fact}(3) \cdot (3+1) = 24$$

阶乘函数通常被表示为 $\text{fact}(x) = x!$ 。 □

原始递归函数是一组可以计算的函数。丘奇-图灵论题证明了这些函数同样必须是可以使用图灵机方法计算的。下面的定理 13.1.3 将会对这一点进行证明。

定理 13.1.3 每个原始递归函数都是图灵可计算的。

证明: 我们已经在 9.2 节中建立了用来执行基本函数计算的图灵机。为了完成证明, 我们要证明图灵可计算的函数在组合和原始递归的操作下是封闭的。前者已经在 9.4 节得到了证明。现在需要证明的是图灵可计算函数在原始递归操作下是封闭的。即, 如果 f 是由图灵可计算函数 h 和 g 原始递归得来的, 那么 f 是图灵可计算的。

设 h 和 g 是图灵可计算的函数, 利用它们进行原始递归而得到的函数 f 定义如下:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$$

因为 h 和 g 是图灵可计算的, 那么就存在标准图灵机 G 和 H 可以计算它们。可以构造一个组合的机器来计算 f 。对于 $f(x_1, \dots, x_n, y)$ 的计算, 是从带上的配置 $B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B y B$ 开始的。

1. 首先在输入的右边紧接着写上一个计数器, 并将它初始化为 0。计数器用来对当前计算中递归变量的值进行计数。接下来在计数器右边写上参数, 从而产生如下的配置:

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{y} B \bar{0} B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B.$$

2. 接下来在 G 上运行带上最后的 n 个值, 产生

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{y} B \bar{0} B \overline{g(x_1, x_2, \dots, x_n)} B$$

G 上的计算产生了 $g(x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n, 0)$ 。

3. 现在带上的格式是

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{y} B \bar{i} B \overline{f(x_1, x_2, \dots, x_n, i)} B$$

如果计数器 i 等于 y , 那么关于 $f(x_1, x_2, \dots, x_n, y)$ 的计算就擦除带上开始的 $n+2$ 个数字, 并将结果放在带上的第一个位置。

4. 如果 $i < y$, 那么为了计算 f 的下一个值, 我们要对带进行重新配置。

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{y} B \bar{i} + 1 B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{i} B \overline{f(x_1, x_2, \dots, x_n, i)} B$$

使用 H 来运行带上最后的 $n+2$ 个值, 产生了

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{y} B \bar{i} + 1 B \overline{h(x_1, x_2, \dots, x_n, i, f(x_1, x_2, \dots, x_n, i))} B,$$

这里带上最右边的值是 $f(x_1, x_2, \dots, x_n, i+1)$ 。计算接下来继续执行第三步中的比较

13.2 一些原始递归函数

如果一个函数可以利用后继函数、零函数和映射函数, 经过有限步地使用组合和原始递归得到, 那么这个函数是原始递归函数。在原始递归函数的定义中, 组合操作允许利用任何已经被证明具有原始递归性质的函数。

原始递归定义是从一些基本的算法函数中得到的。不同于 g 和 h 直接给出细节的信息, 原始递归

的定义是通过参数、递归变量、函数以前的值以及其他的原始递归函数来给出的。需要注意到这里的加函数和乘函数等同于在例 13.1.2 和例 13.1.3 中给出的形式化定义，只是此处省略了中间的步骤。

因为组合和原始递归操作的兼容性，表 13-1 和表 13-2 中的定义都是用函数符号来给出的。表 13-1 的第二列中给出的是函数中缀表示，它将在整章中用作代数表达式。符号“+1”代表了后继操作符。

394

表 13-1 原始递归算术函数

描 述	函 数	定 义
加法	$add(x, y)$	$add(x, 0) = x$
	$x + y$	$add(x, y + 1) = add(x, y) + 1$
乘法	$mult(x, y)$	$mult(x, 0) = 0$
	$x \cdot y$	$mult(x, y + 1) = mult(x, y) + x$
前驱	$pred(y)$	$pred(0) = 0$
		$pred(y + 1) = y$
合适减法	$sub(x, y)$	$sub(x, 0) = x$
	$x - y$	$sub(x, y + 1) = pred(sub(x, y))$
指数运算	$exp(x, y)$	$exp(x, 0) = 1$
	x^y	$exp(x, y + 1) = exp(x, y) \cdot x$

原始递归谓词是值域在 $\{0, 1\}$ 上的原始递归函数。0 和 1 分别代表了假和真。在表 13-2 中的前两个谓词是符号谓词，它说明了参数的符号。当参数是正数的时候，函数 sg 返回真；符号谓词的补表示为 $cosg$ ，当输入为 0 时， $cosg$ 返回真。可以利用算术函数和符号谓词进行组合来得到用于比较输入的二元谓词。

表 13-2 原始递归谓词

描 述	谓 词	定 义
符号	$sg(x)$	$sg(0) = 0$ $sg(y + 1) = 1$
符号的补	$cosg(x)$	$cosg(0) = 1$ $cosg(y + 1) = 0$
小于	$lt(x, y)$	$sg(y \div x)$
大于	$gt(x, y)$	$sg(x \div y)$
等于	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
不等于	$ne(x, y)$	$cosg(eq(x, y))$

395

谓词是表明提出问题正确与否的函数。它的一些逻辑计算，例如取反、合取和析取可以使用算术运算函数以及符号谓词来构造。假设 p_1 和 p_2 是两个原始递归谓词， p_1 和 p_2 的逻辑操作可以按照如右表所示的定义来完成。

谓词	解释
$cosg(p_1)$	p_1 取反
$p_1 \cdot p_2$	p_1 乘 p_2
$sg(p_1 + p_2)$	p_1 加 p_2

使用 $cosg$ 可以对谓词的结果进行取反。这个技术还可以被用来使用谓词 eq 来定义谓词 ne 。析取计算的值是将所有部分谓词的结果进行加和。因为两个谓词都为真，那么经过加和得到的结果是 2，利用 sg 谓词可得到该结果。因为组合中所有的成分都是原始递归的，所以结果谓词也是原始递归的。

例 13.2.1 等于谓词可以用来显示地声明拥有有限参数的函数的值。例如， f 是一个除了对于 0、1 和 2 都返回自身值的函数：

$$f(x) = \begin{cases} 2 & \text{如果 } x=0 \\ 5 & \text{如果 } x=1 \\ 4 & \text{如果 } x=2 \\ x & \text{其他} \end{cases} \quad f(x) = eq(x, 0) \cdot 2 + eq(x, 1) \cdot 5 + eq(x, 2) \cdot 4 + gt(x, 2) \cdot x.$$

函数 f 是原始可递归的，因为它可以被表示为原始递归函数 eq 、 gt 、 \cdot 和 $+$ 的组合。 f 的四个谓词是穷

举的而且是互斥的。即,对于任意的一个自然数,它们里面有且只有一个为真。 f 的值都是由为真的谓词所对应的值来决定的。 \square

在前面的例子中使用穷举而且是互斥的谓词来构造函数的技术,将被用来证明下面的定理

定理 13.2.1 设 g 是原始递归函数, f 为全函数。除了有限个输入的值, f 所有其他的值均等同于 g 。那么 f 也是原始递归的。

证明: 设 g 是原始递归的, 并且定义 f 如下:

$$f(x) = \begin{cases} y_1 & \text{如果 } x = n_1 \\ y_2 & \text{如果 } x = n_2 \\ \vdots & \\ y_k & \text{如果 } x = n_k \\ g(x) & \text{其他.} \end{cases}$$

等于谓词用来声明对于输入为 n_1, \dots, n_k 的 f 的值。对于其他的输入值, $f(x) = g(x)$ 。当 f 的值是由 g 来决定时, 通过下面乘积得到的谓词为真:

$$ne(x, n_1) \cdot ne(x, n_2) \cdot \dots \cdot ne(x, n_k)$$

使用这些谓词, 可以将 f 写为:

$$f(x) = eq(x, n_1) \cdot y_1 + eq(x, n_2) \cdot y_2 + \dots + eq(x, n_k) \cdot y_k \\ + ne(x, n_1) \cdot ne(x, n_2) \cdot \dots \cdot ne(x, n_k) \cdot g(x)$$

因此 f 也是原始递归的。 \blacksquare

变量的顺序是原始递归定义中一个重要的特色。初始的变量都是参数, 最终的变量是递归变量。将组合函数和映射函数结合在一起, 可以在声明原始递归函数中变量的数目和顺序时带来很大的灵活性。在拥有两个变量的函数中, 这种灵活性是通过考虑对变量进行替换来展示的。

定理 13.2.2 设 $g(x, y)$ 是原始递归函数。那么通过下面列举的方式来构造的函数同样也是原始递归的。

i) (增加哑变量) $f(x, y, z_1, z_2, \dots, z_n) = g(x, y)$

ii) (改变变量顺序) $f(x, y) = g(y, x)$

iii) (识别变量) $f(x) = g(x, x)$

证明: 每个函数都是原始递归的, 因为可以它们通过 g 和投影函数使用下面的组合得到:

i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$

ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$

iii) $f = g \circ (p_1^{(1)}, p_1^{(1)})$ \blacksquare

哑变量可以让具有不同变量数目的函数能够进行组合。组合 $h \circ (g_1, g_2)$ 中需要 g_1 和 g_2 具有相同数目的变量。对于两个变量的函数 $f(x, y) = (x \cdot y) + x!$, 其两个构成成分是乘法和阶乘。前面的乘法有两个变量, 而后面的阶乘只需要一个变量。可以在后面的阶乘函数中增加一个哑变量, 从而构造拥有两个变量的函数 $fact'(x, y) = fact(x) = x!$ 。最终我们可以得到 $f = add \circ (mult, fact')$, 所以 f 也是原始递归的。

13.3 有界操作符

对一系列自然数进行求和可以通过重复地使用加法操作来完成。可以将加法和投影函数结合起来, 从而构造一个增加固定数目参数的函数。例如, 原始递归函数

$$add \circ (p_1^{(4)}, add \circ (p_2^{(4)}, add \circ (p_3^{(4)}, p_4^{(4)})))$$

返回的是四个参数的和。当求和的数目是变量的时候, 这个方法是无法使用的。函数

$$f(y) = \sum_{i=0}^y g(i) = g(0) + g(1) + \dots + g(y)$$

参与加法的变量数目是由输入变量 y 来决定的。函数 f 也被称为 g 的有界和 (bounded sum)。变量 i 是和的下标。计算有界和包含了三个操作: 产生被加数, 进行两两加法, 将下标和输入 y 进行比较。

接下来我们要证明求有界和的函数是原始递归的。这里使用的技术可以用来证明重复地使用任意的二元原始递归操作得到的函数同样是原始递归的。

定理 13.3.1 设 $g(x_1, \dots, x_n, y)$ 是原始递归函数。那么下面的函数都是原始递归的:

$$\text{i) (有界和)} f(x_1, \dots, x_n, y) = \sum_{i=0}^y g(x_1, \dots, x_n, i)$$

$$\text{ii) (有界乘积)} f(x_1, \dots, x_n, y) = \prod_{i=0}^y g(x_1, \dots, x_n, i)$$

证明: 有界和

$$\sum_{i=0}^y g(x_1, \dots, x_n, i)$$

398

是把 $g(x_1, \dots, x_n, y)$ 和

$$\sum_{i=0}^{y-1} g(x_1, \dots, x_n, i)$$

相加得到的。

将它转换为原始递归的语言, 我们可以得到

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n, 0)$$

$$f(x_1, \dots, x_n, y+1) = f(x_1, \dots, x_n, y) + g(x_1, \dots, x_n, y+1)$$

有界操作从零开始, 并重复进行, 直到下标达到了参数 y 所声明的值才停止。通过两个可计算函数来确定下标变量的值域, 从而将有界操作进行泛化。函数 l 和 u 分别确定了下标的下界和上界。

定理 13.3.2 设 g 是具有 $n+1$ 个变量的原始递归函数, 并且让 l 和 u 分别表示具有 n 个变量的原始递归函数。那么下面的函数也是原始递归的:

$$\text{i) } f(x_1, \dots, x_n) = \sum_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i)$$

$$\text{ii) } f(x_1, \dots, x_n) = \prod_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i)$$

证明: 因为求和的上界和下界都是由函数 l 和 u 来确定的, 那么有可能下界会比上界更大一些。当这种情况发生的时候, 我们把求和的结果设定为 0。在这些实例中, 谓词

$$gt(l(x_1, \dots, x_n), u(x_1, \dots, x_n))$$

始终为真。

如果产生的下界小于或者等于上界, 那么求和就应该从 $l(x_1, \dots, x_n)$ 开始, 当下标到达 $u(x_1, \dots, x_n)$ 时停止。设 g' 是原始递归函数

$$g'(x_1, \dots, x_n, y) = g(x_1, \dots, x_n, y + l(x_1, \dots, x_n))$$

g' 的值是由 g 和 $l(x_1, \dots, x_n)$ 得到的。

$$g'(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n, l(x_1, \dots, x_n))$$

$$g'(x_1, \dots, x_n, 1) = g(x_1, \dots, x_n, 1 + l(x_1, \dots, x_n))$$

\vdots

$$g'(x_1, \dots, x_n, y) = g(x_1, \dots, x_n, y + l(x_1, \dots, x_n))$$

399

根据定理 13.3.1, 下面的函数

$$\begin{aligned} f'(x_1, \dots, x_n, y) &= \sum_{i=0}^y g'(x_1, \dots, x_n, i) \\ &= \sum_{i=l(x_1, \dots, x_n)}^{y+l(x_1, \dots, x_n)} g(x_1, \dots, x_n, i) \end{aligned}$$

也是原始递归的。泛化的有界和可以通过将函数 l 和 u 以及 f' 进行组合来得到:

$$f'(x_1, \dots, x_n, (u(x_1, \dots, x_n) \div l(x_1, \dots, x_n))) = \sum_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i).$$

将这个函数与比较上界下界的谓词进行乘积, 可以保证当下界超过上界的时候, 有界和函数总是返回默认值。因此

$$f(x_1, \dots, x_n) = \text{cosg}(gt(l(x_1, \dots, x_n), u(x_1, \dots, x_n))) \\ \cdot f'(x_1, \dots, x_n, (u(x_1, \dots, x_n) \div l(x_1, \dots, x_n))).$$

因为式子里面每个组成的函数都是原始递归的, 所以 f 同样也是原始递归的。

可以使用类似的过程来证明泛化的有界乘积也是原始递归的。当下界超过上界的时候, 有界乘积函数总是返回 0。

谓词 p 返回的值表明了输入是否满足属性 p 。对于固定的值 x_1, \dots, x_n , 我们可以使用

$$\mu z[p(x_1, \dots, x_n, z)]$$

来表示满足 $p(x_1, \dots, x_n, z) = 1$ 的最小自然数 z 。 $\mu z[p(x_1, \dots, x_n, z)]$ 被称为“满足 $p(x_1, \dots, x_n, z)$ 的最小 z ”。这种构造方式可以称为 p 的最小化。 μz 被称为 μ 操作符。 $n+1$ 个变量谓词的最小化定义了一个 n 元的函数:

$$f(x_1, \dots, x_n) = \mu z[p(x_1, \dots, x_n, z)].$$

对于最小化, 一个直观的解释是它在自然数中执行了查询。在开始的时候, 变量 z 被设定为零。查询对自然数进行顺序的检查, 直到发现有 z 满足 $p(x_1, \dots, x_n, z) = 1$ 。

不幸的是, 由原始递归谓词得到的最小化函数并不需要是原始递归的。实际上, 这个函数甚至不必是完全的。考虑下面的函数

$$f(x) = \mu z[eq(x, z \cdot z)].$$

利用最小化的特点进行查询, f 首先查找第一个满足 $z^2 = x$ 的 z 。如果 x 的平方根是个整数, 那么 $f(x)$ 就返回 x 的平方根。否则, f 便没有定义。

通过对最小化的值域进行限制, 我们可以得到有界的最小化操作符。 $n+1$ 个变量的谓词定义了一个具有 $n+1$ 个变量的函数

$$f(x_1, \dots, x_n, y) = \mu z[p(x_1, \dots, x_n, z)]$$

$$= \begin{cases} z & \text{如果 } p(x_1, \dots, x_n, i) = 0, \text{ 其中 } 0 \leq i < z \leq y \\ & \text{和 } p(x_1, \dots, x_n, z) = 1 \\ y+1 & \text{其他。} \end{cases}$$

有界 μ 操作符返回的是满足 $p(x_1, \dots, x_n, z) = 1$ 并且小于或者等于 y 的第一个自然数 z 。如果这样的值不存在, 那么它就返回默认的 $y+1$ 。通过将查询的范围缩小至 0 和 y , 整个函数的完整性便可以得到保证。

$$f(x_1, \dots, x_n, y) = \mu^y z[p(x_1, \dots, x_n, z)].$$

实际上, 当谓词是原始递归的时候, 有界最小化操作符定义的是一个原始递归函数。

定理 13.3.3 设 $f(x_1, \dots, x_n, y)$ 是原始递归谓词, 那么函数

$$f(x_1, \dots, x_n, y) = \mu^y z[p(x_1, \dots, x_n, z)]$$

也是原始递归的。

证明: 证明给出了一个两元谓词, 并可以很容易地泛化到具有 n 个变量的谓词上去。我们定义这样一个谓词

$$g(x, y) = \begin{cases} 1 & \text{如果 } p(x, i) = 0, \text{ 其中 } 0 \leq i \leq y \\ 0 & \text{其他} \end{cases} \\ = \prod_{i=0}^y \text{cosg}(p(x, i)).$$

[401] 这个谓词是原始递归的, 因为它是原始递归谓词 $\text{cosg} \circ p$ 的有界乘积。

有界和的谓词 p 产生了有界的 μ 操作符。为了解释如何使用 p 来构造最小化的操作符, 我们使用参数 n 来构造一个有两个变量的谓词 p , 具体的值在下面给出:

$$\begin{array}{lll}
 p(n,0)=0 & g(n,0)=1 & \sum_{i=0}^0 g(n,i)=1 \\
 p(n,1)=0 & g(n,1)=1 & \sum_{i=0}^1 g(n,i)=2 \\
 p(n,2)=0 & g(n,2)=1 & \sum_{i=0}^2 g(n,i)=3 \\
 p(n,3)=1 & g(n,3)=0 & \sum_{i=0}^3 g(n,i)=3 \\
 p(n,4)=0 & g(n,4)=0 & \sum_{i=0}^4 g(n,i)=3 \\
 p(n,5)=1 & g(n,5)=0 & \sum_{i=0}^5 g(n,i)=3 \\
 \vdots & \vdots & \vdots
 \end{array}$$

g 的值始终都为 1, 直到出现第一个满足 $p(n,z)=1$ 的 z 。接下来 g 所有的值均为 0。有界和加上 f 由 g 产生的值。因此,

$$\sum_{i=0}^y g(n,i) = \begin{cases} y+1 & \text{如果 } z > y \\ z & \text{其他。} \end{cases}$$

第一个条件同样包含了没有 z 满足 $p(n,z)=1$ 的可能性。在这种情况下, 我们将直接返回默认值。

利用前面的证明, 我们可以看到有界的原始递归谓词 p 是由函数

$$f(x,y) = \mu^y z[p(x,z)] = \sum_{i=0}^y g(x,i),$$

给出的, 因此也是原始递归的。

有界最小化 $f(y) = \mu^y z[p(x,z)]$ 可以被视为在值域 0 到 y 之间寻找第一个使 p 为真的值。例 13.3.1 表明了最小化同样可以被用于在一个子值域中寻找第一个满足条件的值, 或者在指定的值域中寻找满足 p 的最大的 z 。

[402]

例 13.3.1 如果 $p(x,z)$ 是原始递归的谓词。那么

- i) $f_1(x,y_0,y)$ = 在值域 $[y_0,y]$ 内第一个使 $p(x,z)$ 为真的值,
- ii) $f_2(x,y)$ = 在值域 $[0,y]$ 内第二个使 $p(x,z)$ 为真的值,
- iii) $f_3(x,y)$ = 在值域 $[0,y]$ 内使 $p(x,z)$ 为真的最大值。

这些函数同样也是原始递归的。对于这些函数, 如果没有 z 能够满足规约的条件, 那么默认的值是 $y+1$ 。

为了证明 f_1 是原始递归的, 我们使用原始递归的函数 ge (大于或等于) 来保证函数值的下界。当 $p(x,z)$ 为真, 而且 z 大于或者等于 y_0 的时候, 谓词 $p(x,z) \cdot ge(z,y_0)$ 为真。有界最小化

$$f_1(x,y_0,y) = \mu^y z[p(x,z) \cdot ge(z,y_0)],$$

能够返回在值域 $[y_0,y]$ 内第一个使 $p(x,z)$ 为真的值。

最小化 $\mu^y z[p(x,z)]$ 是在值域 $[0,y]$ 内第一个使 $p(x,z)$ 为真的值。而第二个使 $p(x,z)$ 为真的值则是满足 p 而且大于 $\mu^y z[p(x,z)]$ 的第一个值。使用前面的技术, 函数

$$f_2(x,y) = \mu^y z[p(x,z) \cdot gt(z, \mu^y z[p(x,z)])]$$

则返回了在值域 $[0,y]$ 内第二个使 $p(x,z)$ 为真的值。

在值域 $[0,y]$ 内寻找最大值需要在 $y, y-1, y-2, \dots, 1, 0$ 中顺序进行查找。有界最小化 $\mu^y z[p(x,y-z)]$ 按照指定的顺序来进行检查。当 $z=0$ 时, 就测试 $p(x,y)$; 当 $z=1$ 时, 就测试 $p(x,y-1)$, 如此继

续下去。函数 $f(x, y) = y - \mu z[p(x, y \div z)]$ 返回了满足 p 而且小于或者等于 y 的最大值。然而, 当这样的值不存在的时候, f 的值是 $y \div (y + 1) = 0$ 。这里要进行比较从而返回正确的默认值。函数

$$f_3(x, y) = eq(y + 1, \mu z[p(x, z)]) \cdot (y + 1) + neq(y + 1, \mu z[p(x, z)]) \cdot f(x, y)$$

中第一个条件能够在值域 $[0, y]$ 内没有值满足 p 时返回默认的 $y + 1$ 。否则, 就返回最大的值。□

利用函数 u 来计算查询的上界便可以泛化有界最小化。如果 u 是原始递归的, 那么相应的函数也是。证明过程和定理 13.3.2 类似, 留作练习。

定理 13.3.4 设 p 是具有 $n + 1$ 个变量的原始递归谓词, 设 u 是具有 n 个变量的原始递归函数, 那么函数

$$f(x_1, \dots, x_n) = \mu z^{u(x_1, \dots, x_n)} [p(x_1, \dots, x_n, z)]$$

也是原始递归的。

13.4 除法函数

整数除法 (div) 的基本操作并不是全函数。函数 $div(x, y)$ 返回的是使用 x 除以 y 所得的整数部分, 而且要求第二个参数不能为 0。而当 y 为 0 时, 函数是没有意义的。因为所有的原始递归函数都是全函数, 所以 div 函数不是原始递归的。如果我们定义当 y 为 0 时, 函数返回一个默认值, 便可以得到一个原始递归的函数 quo :

$$quo(x, y) = \begin{cases} 0 & \text{如果 } y = 0 \\ div(x, y) & \text{其他。} \end{cases}$$

使用乘法的原始递归操作可以构造除法函数 quo 。对于每个非零的数, $quo(x, y) = z$ 意味着 z 满足 $z \cdot y \leq x < (z + 1) \cdot y$ 。即, $quo(x, y)$ 返回的是满足 $(z + 1) \cdot y$ 大于 x 的最小的 z 。寻找 z 的过程在 z 到达 x 之前就会成功, 因为 $(z + 1) \cdot y$ 比 x 要大。函数

$$\mu z[gt((z + 1) \cdot y, x)]$$

确定了当除法 div 存在定义时, x 和 y 的商。默认值是由将最小化和 $sg(y)$ 相乘得到的。因此, 在

$$quo(x, y) = sg(y) \cdot \mu z[gt((z + 1) \cdot y, x)],$$

中, 边界是由原始递归函数 p_1^2 决定的。前面的定义证明了 quo 是原始递归的, 因为它具有定理 13.3.4 中所描述的形式。

求商的函数可以用来定义表 13-3 中一系列包括和除法相关的函数和谓词。函数 rem 返回了当除法有定义时, x 和 y 执行除法之后的余数。否则, $rem(x, 0) = x$ 。谓词 $divides$ 定义如下:

$$divides(x, y) = \begin{cases} 1 & \text{若 } x > 0, y > 0, \text{ 且 } y \text{ 是 } x \text{ 的因子} \\ 0 & \text{其他} \end{cases}$$

当 x 可以被 y 整除时, 谓词 $divides$ 为真。通常情况下, 零被认为不可以整除任何数的。在表 13-3 对 $divides$ 的定义中, 它和 $sg(x)$ 相乘确保了这个条件。剩下的函数保证了 $divides(x, 0) = 0$ 。

表 13-3 原始递归除函数

描 述	函 数	定 义
求商	$quo(x, y)$	$sg(y) \cdot \mu z[gt((z + 1) \cdot y, x)]$
求余	$rem(x, y)$	$x - (y \cdot quo(x, y))$
整除	$divides(x, y)$	$eq(rem(x, y), 0) \cdot sg(x)$
整除数个数	$ndivisors(s, y)$	$\sum_{i=0}^s divides(x, i)$
素数	$prime(x)$	$eq(ndivisors(x), 2)$

泛化的有界和可以用来计算一个数所拥有的整除数的个数。和的上界是通过原始递归函数 $p_1^{(1)}$ 操作输入得到的。这个边界是可以满足的, 因为没有大于 x 的数可以作为 x 的整除数。素数是仅能被 1 和它自身整除的数, 谓词 $prime$ 仅仅检查数的整除数的个数是否为 2。

谓词 *prime* 和有界最小化可以用来构造原始递归函数 pn , 从而可以将这些素数枚举出来 $pn(i)$ 的值是第 i 个素数。因此, $pn(0) = 2, pn(1) = 3, pn(2) = 5, pn(3) = 7, \dots$ 。这里第 $x+1$ 个素数是大于 $pn(x)$ 的第一个素数。有界最小化比较适合执行这类查询。为了使用 μ 操作符, 我们必须为最小化确定上界。利用定理 13.3.4, 这个界限可以使用输入 x 计算得到。

引理 13.4.1 设 $pn(x)$ 的值是第 i 个素数, 那么 $pn(x+1) \leq pn(x)! + 1$ 。

证明: 对于 $i = 0, 1, \dots, x$ 的每个素数 $pn(i)$ 均能够整除 $pn(x)!$ 。既然一个素数不会被连续的两个数整除, 所以要么 $pn(x)! + 1$ 是素数, 要么使用素数进行分解包含了 $pn(0), pn(1), \dots, pn(x)$ 之外的素数。无论哪种情况, $pn(x+1) \leq pn(x)! + 1$ 。 ■

上面的引理提供的边界可以通过原始递归函数 $fact(x) + 1$ 来计算。求第 x 个素数的函数可以通过如下的原始递归得到的:

$$pn(0) = 2$$

$$pn(x+1) = \mu z^{fact(pn(x))+1} [prime(z) \cdot gt(z, pn(x))].$$

我们可以映射一下原始递归函数家族和图灵可计算性之间的关系。根据定理 13.1.3, 每个原始递归函数都是图灵可计算的。设计能够执行 pn 和 $ndivisors$ 这样函数的图灵机需要很多的状态和复杂的转换函数, 而使用宏观的方法来看计算的问题就可以很容易地证明这些函数是可计算的。借助于此, 不需要构造复杂的图灵机, 我们已经证明了很多有用的函数和谓词都是图灵可计算的。

405

13.5 歌德尔数字和串值递归

很多涉及自然数的通用计算函数都不是数论函数。例如将一个序列的数进行排序的函数可以返回一个有序的序列, 而不是单个的数。高效的排序方法有很多。在这里我们可以通过引入原始递归操作来完成类似的操作。操作的核心是将序列中的数编码为单个的值。编码的机制将自然数惟一地分解为一系列素数的乘积。这种编码被称为歌德尔 (Gödel) 数字, 这是根据发明这项技术的德国逻辑学家 Kurt Gödel 来命名的。

n 个自然数 x_0, x_1, \dots, x_{n-1} 的序列可以被编码为

$$pn(0)^{x_0+1} \cdot pn(1)^{x_1+1} \cdot \dots \cdot pn(n)^{x_{n-1}+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \dots \cdot pn(n)^{x_{n-1}+1}.$$

因为这组数计数从 0 开始, 那么序列中的 n 个元素就被编号为 $0, 1, \dots, n-1$ 。使用歌德尔数字表示的序列有: 经过编码的长度为 n 的序列是最小的 n 个素数的幂指数乘积。在指数上我们使用的是 $x_i + 1$, 这样可以保证即使在 x_i 为零的时候, $pn(i)$ 也会在编码中出现。

能够完成将固定数目的输入进行编码的函数可以直接从歌德尔数字的定义中推导出来, 我们设:

序列	编码
1, 2	$2^2 3^3 = 108$
0, 1, 3	$2^1 3^2 5^4 = 11250$
0, 1, 0, 1	$2^1 3^2 5^1 7^2 = 4410$

$$gn_n(x_0, \dots, x_n) = pn(0)^{x_0+1} \cdot \dots \cdot pn(n)^{x_n+1} = \prod_{i=0}^n pn(i)^{x_i+1}$$

是能够对具有 $n+1$ 个变量 x_0, x_1, \dots, x_n 的序列进行编码的函数。函数 gn_n 可以用来对有序的 n 元组进行编码。和有序对 $[x_0, x_1]$ 对应的歌德尔数字是 $gn_1(x_0, x_1)$ 。

406

一个解码函数可以用来从编码序列中取出相应的数据。函数

$$dec(i, x) = \mu z [\cosg(divides(x, pn(i)^{z+1}))] \div 1$$

返回在歌德尔数字 x 中的第 i 个编码元素。有界 μ 操作符可以用来在 x 的素数分解中查找 $pn(i)$ 的指数。这个最小化操作返回的是第一个不能使 $pn(i)^{z+1}$ 整除 x 的 z 。编码序列中的第 i 个元素比编码中 $pn(i)$ 的指数小。对于每个不在 x 的素数分解中出现的 $pn(i)$, 解码函数 $dec(i, x)$ 返回的是 0。

当计算需要得到前面的 n 个数字时, 那么可以利用歌德尔编码函数 gn_{n-1} 对这些值进行编码。当需要的时候, 编码的值可以从序列中检索出来。

例 13.5.1 斐波纳契 (Fibonacci) 数列定义为如下的序列: $0, 1, 1, 2, 3, 5, 8, 13, \dots$, 这里面每个元素都是前面两个元素的和。函数

$$f(0) = 0$$

$$f(1) = 1$$

$$f(y+1) = f(y) + f(y-1) \text{ 对于 } y > 1$$

可以产生斐波纳契数列。这里的定义并不是原始递归的, 因为 $f(y+1)$ 的计算利用了 $f(y)$ 和 $f(y-1)$ 。为了表明斐波纳契数列可以利用原始递归函数来生成, 我们可以利用歌德尔数字来将两个值存储为 gn_1 , 并可以使用额外的函数 h 来对有序对中的 $f(y-1)$ 和 $f(y)$ 进行编码:

$$h(0) = gn_1(0, 1) = 2^1 3^2 = 18$$

$$h(y+1) = gn_1(dec(1, h(y)), dec(0, h(y)) + dec(1, h(y))).$$

h 的初始值是编码的 $[f(0), f(1)]$ 。 $h(y+1)$ 的计算从生成下面的有序对开始

$$[dec(1, h(y)), dec(0, h(y)) + dec(1, h(y))] = [f(y), f(y-1) + f(y)].$$

使用 gn_1 来对结果进行编码就完成了对 $h(y+1)$ 的计算。这个过程会构造 $[f(0), f(1)], [f(1), f(2)], [f(2), f(3)], \dots$ 等有序对序列的歌德尔数字。原始递归函数 $f(y) = dec(0, h(y))$ 能够从有序对的第一部分中提取出斐波纳契数列。□

歌德尔数字编码函数 gn_1 可以对固定数目的参数进行编码。同样, 也可以构造出参数个数是由计算而得的歌德尔数字编码函数。具体的构造方法与构造计算有界和以及有界乘积的操作类似。对于只有一个变量的原始递归函数 f , 可以根据它的输入 $0, 1, \dots, n$ 定义一个长度为 $n+1$ 的序列 $f(0), f(1), \dots, f(n)$ 。使用有界积, 歌德尔数字编码函数

$$gn_f(x_1, \dots, x_n, y) = \prod_{i=0}^y pn(i)^{f(i)+1}$$

可以对 f 的前 $y+1$ 个值进行编码。函数 f 和编码函数 gn_f 之间的关系将在定理 13.5.1 中给出。

定理 13.5.1 设 f 是有 $n+1$ 个变量的函数, 编码函数 gn_f 根据 f 来定义。那么 f 是原始递归的当且仅当 gn_f 是原始递归的。

证明: 如果 $f(x_1, \dots, x_n, y)$ 是原始递归的, 那么有界积

$$gn_f(x_1, \dots, x_n, y) = \prod_{i=0}^y pn(i)^{f(x_1, \dots, x_n, i)+1}$$

能够产生歌德尔数字编码函数。在另一方面, 解码函数可以用来从歌德尔数字编码中恢复 f 的数值。

$$f(x_1, \dots, x_n, y) = dec(y, gn_f(x_1, \dots, x_n, y)).$$

因此, f 是原始递归的当且仅当 gn_f 是原始递归的。■

正是考虑到原始递归函数具有直观可计算性, 我们才引入了这个概念。在原始递归的定义中, 我们允许利用递归变量前面的计算结果来执行当前的计算。对于下面的函数:

$$f(0) = 1$$

$$f(1) = f(0) \cdot 1 = 1$$

$$f(2) = f(0) \cdot 2 + f(1) \cdot 1 = 3$$

$$f(3) = f(0) \cdot 3 + f(1) \cdot 2 + f(2) \cdot 1 = 8$$

$$f(4) = f(0) \cdot 4 + f(1) \cdot 3 + f(2) \cdot 2 + f(3) \cdot 1 = 21$$

⋮

函数 f 可以表示为

$$f(0) = 1$$

$$f(y+1) = \sum_{i=0}^y f(i) \cdot (y+1-i).$$

给出的定义并不是原始递归的, 因为 $f(y+1)$ 的计算使用了前面所有的计算结果。但是, 这个函数直观上却是可计算的, 因为函数定义本身给出了如何执行计算的算法。

当递归变量 $y+1$ 对应的值是根据 $f(0), f(1), \dots, f(y)$ 而定义的时候, 我们称函数 f 是串值递归定义的。对于使用串值递归定义的函数来说, 当计算函数值的时候往往需要使用递归变量中的每个输入

407

408

值。在前面的例子中, $f(2)$ 的计算需要 $f(0)$ 和 $f(1)$, 而 $f(4)$ 的计算则需要 $f(0)$ 、 $f(1)$ 、 $f(2)$ 和 $f(3)$ 。单个函数无法直接从前面的值来计算 $f(2)$ 和 $f(4)$, 因为一个函数要求参数的个数是固定的。

无论递归变量 $y+1$ 的值是多少, 前面的结果均可以使用歌德尔数字编码函数来进行编码。这一点为给串值递归进行形式化的定义提供了基础。

定义 13.5.2 设 g 和 h 分别是 $n+2$ 个变量的数论全函数。具有 $n+1$ 个变量的函数 f 定义如下:

- i) $f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$
- ii) $f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, gn_f(x_1, \dots, x_n, y))$

这个函数被称为从 g 和 h 利用计值过程递归得到的。

定理 13.5.3 如果具有 $n+1$ 个变量的函数 f 是由原始递归函数 g 和 h 利用计值过程递归得到的, 那么 f 也是原始递归的。

证明: 我们从原始递归函数 g 和 h 开始来定义 gn_f 。

$$\begin{aligned} gn_f(x_1, \dots, x_n, 0) &= 2^{f(x_1, \dots, x_n, 0)+1} \\ &= 2^{g(x_1, \dots, x_n)+1} \\ gn_f(x_1, \dots, x_n, y+1) &= gn_f(x_1, \dots, x_n, y) \cdot pn(y+1)^{f(x_1, \dots, x_n, y+1)+1} \\ &= gn_f(x_1, \dots, x_n, y) \cdot pn(y+1)^{n+1+g(x_1, \dots, x_n, y)+1} \end{aligned}$$

对于 $gn_f(x_1, \dots, x_n, y+1)$ 的计算, 我们使用了:

- i) 参数 x_0, \dots, x_n ,
- ii) 递归变量的前一个值 y ,
- iii) gn_f 的前一个值 $gn_f(x_1, \dots, x_n, y)$,
- iv) 原始递归函数 h 、 pn 、 \cdot 、 $+$ 和指数函数。

因此, gn_f 的计算是原始递归的。根据定理 13.5.1, 我们可以得出 f 是原始递归的。 ■

409

从机器的角度来看, 歌德尔数字编码函数可以使得计算拥有不受限制的内存存储能力。单个的歌德尔数字编码可以存储前面任何结果的值。歌德尔数字可以对计算 $f(x_0, \dots, x_n, y+1)$ 所需的 $f(x_0, \dots, x_n, 0)$ 、 $f(x_0, \dots, x_n, 1)$ 和 $f(x_0, \dots, x_n, y)$ 都进行编码。解码函数则在记忆和计算之间提供了联系。当需要使用存储的值来进行计算时, 解码函数能够满足需求。

例 13.5.2 设 h 是原始递归函数

$$h(x, y) = \sum_{i=0}^x dec(i, y) \cdot (x+1-i).$$

那么在前面计值计算中定义的函数 f , 可以在 h 上进行计值过程递归来定义

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= h(y, gn_f(y)) = \sum_{i=0}^y dec(i, gn_f(y)) \cdot (y+1-i) \\ &= \sum_{i=0}^y f(i) \cdot (y+1-i) \end{aligned}$$

□

13.6 可计算部分函数

原始递归函数是指直觉上可计算的函数家族。我们已经证明了所有原始递归函数都是全函数。相应地, 是不是所有的可计算的全函数都是原始递归的呢? 而且, 我们是否应该将对可计算性的分析限制到全函数的范畴上呢? 在本节中, 我们将对这两个问题给出否定的证明。

我们将使用对角化的证明方法来说明存在一个可计算的全函数, 而它却不是原始递归的。第一步是证明原始递归函数的语法结构允许对它们进行有效地枚举。而且, 可以将原始递归函数列举出来, 又使得我们可以构造出与列表中存在的所有函数均不同的新函数。

定理 13.6.1 原始递归函数的集合是可以有效计算的数论函数集合的真子集。

证明: 原始递归函数可以表示为字母表 $\Sigma = \{s, p, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), o, ;, <, >\}$ 上的字

[410] 字符串。基本函数 s, z 和 $p_i(j)$ 表示为 $\langle s \rangle, \langle z \rangle$ 和 $\langle p_i(j) \rangle$ 。组合函数 $h \circ (g_1, \dots, g_n)$ 可以被表示为 $\langle \langle h \rangle \circ \langle \langle g_1 \rangle, \dots, \langle g_n \rangle \rangle \rangle$ ，这里的 $\langle h \rangle$ 和 $\langle g_i \rangle$ 都是成分函数的表示。利用 g 和 h 使用原始递归得到的函数表示为 $\langle \langle g \rangle : \langle h \rangle \rangle$ 。

在 Σ^* 中的字符串可以按照长度来依次生成：首先是空串，然后是长度为 1 的字符串，接下来是长度为 2 的字符串，依此类推。我们可以很直观地设计一个机械的过程来判断字符串是否代表了一个原始递归函数。通过重复地生成字符串并判断它是否是一个函数语法的正确表示，便可以完成对原始递归函数的枚举。第一个满足条件的字符串可以表示为 f_0 ，第二个表示为 f_1 ，依此类推。基于同样的方式，我们可以枚举出所有只拥有一个变量的原始递归函数。这一点可以通过在先前生成的列表中删除所有参数个数大于 1 的函数来完成。因此，可以将序列表示为： $f_0^{(1)}, f_1^{(1)}, f_2^{(1)}, \dots$

拥有一个变量的全函数

$$g(i) = f_i^{(1)}(i) + 1$$

是可计算的。因为可以将所有拥有一个变量的原始递归函数枚举出来，由此便可得出 g 的可计算性。 $g(i)$ 的值可以通过下面的步骤得到：

- i) 确定第 i 个拥有一个变量的原始递归函数 $f_i^{(1)}$ ，
- ii) 计算 $f_i^{(1)}(i)$ ，并且
- iii) 将 $f_i^{(1)}(i)$ 加一。

因为每一步都是可计算的，所以我们总结得出 g 是可计算的。根据对角化证明原理，对于任意 i ，

$$g(i) \neq f_i^{(1)}(i)$$

因此， g 是可计算的全函数，但却不是原始递归的。 ■

定理 13.6.1 使用对角化方法证明了存在可计算但却不是原始递归的函数。这一点也可以通过直接构造可计算而不是原始递归的函数来完成。有两个变量的数论函数，通常被称为阿克曼函数 (ackermann's function)，就是这样一个函数，它的定义如下：

- i) $A(0, y) = y + 1$
- ii) $A(x + 1, 0) = A(x, 1)$
- iii) $A(x + 1, y + 1) = A(x, A(x + 1, y))$

阿克曼函数的值是递归定义的，它的基础是等式 (i)。通过在 x 上的推导，可以知道对于每一对输入的值，它的结果是惟一的（留作练习 22）。例 13.6.1 中的计算证明了阿克曼函数的可计算性。

[411]

例 13.6.1 $A(1, 1)$ 和 $A(3, 0)$ 的值可以从阿克曼函数的定义得到。下面右侧的列给出了替换的步骤：

$$\begin{aligned}
 \text{a) } A(1, 1) &= A(0, A(1, 0)) && \text{(iii)} \\
 &= A(0, A(0, 1)) && \text{(ii)} \\
 &= A(0, 2) && \text{(i)} \\
 &= 3 \\
 \text{b) } A(2, 1) &= A(1, A(2, 0)) && \text{(iii)} \\
 &= A(1, A(1, 1)) && \text{(ii)} \\
 &= A(1, 3) && \text{(a)} \\
 &= A(0, A(1, 2)) && \text{(iii)} \\
 &= A(0, A(0, A(1, 1))) && \text{(iii)} \\
 &= A(0, A(0, 3)) && \text{(a)} \\
 &= A(0, 4) && \text{(i)} \\
 &= 5 && \text{(i)}
 \end{aligned}$$

□

阿克曼函数的值展示出了很快的增长速度。把阿克曼函数的第一个变量固定，得到的是仅有一个变量的函数：

$$A(1, y) = y + 2$$

$$A(2, y) = 2y + 3$$

$$A(3, y) = 2^{y+3} - 3$$

$$A(4, y) = 2^{2^{y+3}} - 3.$$

在 $A(4, y)$ 中, 指数链上 2 的个数是 y 。例如, $A(4, 0) = 16 - 3$, $A(4, 1) = 2^{16} - 3$ 而且 $A(4, 2) = 2^{2^{16}} - 3$ 。阿克曼函数中的第一个变量决定了函数值增长的速度。下面我们将介绍关于阿克曼函数和原始递归函数增长速度比较的定理。在这里我们不给出定理的证明过程。

定理 13.6.2 对于每个只有一个变量的原始递归函数 f , 存在 $i \in \mathbb{N}$ 使得 $f(i) < A(i, i)$ 。

很明显, 单个变量的函数 $A(i, i)$ 不是原始递归的。这也就意味着阿克曼函数不是原始递归的。假设它的话, 那么可以通过 $A \circ (p^{-1}, p_i^{-1})$ 得到 $A(i, i)$, 若如此, 则 $A(i, i)$ 也将是原始递归的。 [412]

是不是可以在原始递归函数的集合中增加一些基本函数或者是额外的操作, 可以使它包含所有的可计算全函数? 不幸的是, 答案是否定的。无论全函数集合是否是可计算的, 定理 13.6.1 中的对角线证明表明了我们无法将所有可计算的全函数枚举出来。因此, 我们可以总结得出可计算函数是无法有效产生的, 或者是存在不可计算的全函数。如果我们接受了后面的结论, 那么在对角线证明中出现的矛盾就会消失。我们不接受 g 是 f_i 中的一员是因为 $g(i) \neq f_i^{-1}(i)$ 。如果 $f_i^{-1}(i) \uparrow$, 那么 $g(i) = f_i^{-1}(i) + 1$ 同样也是没有定义的。如果我们希望能够有效地枚举所有的可计算函数, 那就应该在枚举中包含部分函数。

接下来我们考虑部分函数的可计算性。因为组合和原始递归可以保持全函数的性质, 所以需要增加额外的操作符号, 从而能够从基本函数产生部分函数。最小化操作可以被形式化地描述为查询过程: 在要检查的自然数上设定一个界限, 可以保证有界最小化操作产生的是全函数, 无界最小化可以通过在查询的时候, 不在自然数上设置上界来实现。函数

$$f(x) = \mu z[eq(x, z \cdot z)]$$

定义了无界最小化操作。当 x 的平方根是整数的时候, 无界最小化返回它的平方根。否则, 查询一直寻找第一个满足条件的整数, 直到无穷大。虽然 eq 是全函数, 但函数 f 却不是, 例如 $f(3) \uparrow$ 。如果对于输入 x 的查询没能返回值, 那么利用无界最小化定义的函数对于 x 就是没有定义的。

随着部分函数的引入, 我们不得不重新检查组合和原始递归操作。在定义组合的时候, 需要考虑输入没有定义的可能性。对于输入 x_1, \dots, x_k , 如果下列的条件中有一个为真, 那么函数 $h \circ (g_1, \dots, g_n)$ 便是无定义的:

i) 对于某个 $1 \leq i \leq n, g_i(x_1, \dots, x_k) \uparrow$, 或者

ii) 对于所有的 $1 \leq i \leq n, g_i(x_1, \dots, x_k) \downarrow$, 并且 $h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)) \uparrow$ 。

任何在 g_i 上没有定义的值都会传播到组合函数中。

原始递归操作需要参与定义的函数 f 和 g 都是全函数。这里可以将原始递归的限制放宽, 允许使用部分函数。 f 是通过部分函数 g 和 h 进行原子递归操作来定义的。

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$$

使用原始递归的定义计算函数值是一个迭代的过程。对于函数 f , 递归变量 y 的值仅当下列条件满足的时候才有定义: [413]

i) $f(x_1, \dots, x_n, 0) \downarrow$, 如果 $g(x_1, \dots, x_n) \downarrow$

ii) $f(x_1, \dots, x_n, y+1) \downarrow$, 如果对于 $0 \leq i \leq y, f(x_1, \dots, x_n, i) \downarrow$, 并且 $h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \downarrow$ 。

递归变量未定义的值使得 f 对于相应的递归变量后所有的值都没有定义。

利用部分函数定义中的协定, 可以使用组合、原始递归和无界最小化操作来定义可计算部分函数家族。

定义 13.6.3 μ -递归函数 (μ -recursive function) 家族定义如下:

i) 后继函数、零函数和投影函数都是 μ -递归函数。

ii) 如果 h 是一个有 n 个输入变量的 μ -递归函数, 而 g_1, \dots, g_n 是有 k 个输入变量的 μ -递归函数, 那么 $f = h \circ (g_1, \dots, g_n)$ 是 μ -递归函数。

iii) 如果 g 和 h 分别是有 n 和 $n+2$ 个变量的 μ -递归函数, 那么由 g 和 h 利用原始递归得到的函数 f 也是 μ -递归函数。

iv) 如果 $p(x_1, \dots, x_n, y)$ 是一个 μ 递归的全谓词, 那么 $f = \mu z[p(x_1, \dots, x_n, z)]$ 是 μ -递归函数。

v) 如果一个函数可以从条件(i)中的函数出发、通过有限次使用(ii)(iii)(iv)中的规则得到, 那么这个函数就是 μ -递归函数。

条件(i)和(ii)(iii)意味着所有的原始递归函数都是 μ -递归函数。注意定义中无界最小化并不是对于所有的谓词都有定义的, 而只是对 μ -递归的全谓词有定义。

图灵机中可计算性的含义自然包含了部分函数。图灵机能够计算一个部分的数论函数 f , 如果 f 满足:

i) 当 $f(x_1, \dots, x_n) \downarrow$ 时, 计算以 $f(x_1, \dots, x_n)$ 为结果停机;

ii) 当 $f(x_1, \dots, x_n) \uparrow$ 时, 计算不停机。

当结果存在时, 图灵机总会计算出函数的值。否则, 计算会无限地进行下去。

接下来我们将在 μ -递归函数和图灵可计算函数之间建立联系。过程的第一步是表明所有 μ -递归函数都是图灵可计算的, 这一点并不让人吃惊, 因为这仅仅是将定理 13.1.3 中的结论进行了延伸。

[414] **定理 13.6.4** 每个 μ -递归函数都是图灵可计算的。

证明: 因为基本函数都是图灵可计算的, 所以只需要表明图灵可计算的部分函数在组合、原始递归和无界最小化操作下是闭合的就可以了。在定理 9.4.3 和定理 13.1.3 中, 我们已经证明了图灵可计算的全函数在组合、原始递归下面都是闭合的。这些操作同样也为部分函数构造了闭包。在计算过程中, 在函数某成分计算过程中出现的未定义值将会使整个计算无限制的进行下去。

下面只需要证明图灵可计算的全谓词在无界最小化操作下是图灵可计算的即可。设 $f(x_1, \dots, x_n) = \mu z[p(x_1, \dots, x_n, z)]$, 这里 $p(x_1, \dots, x_n, y)$ 是图灵可计算的全谓词。计算 f 的图灵机可以根据执行谓词 p 的图灵机 P 进行构造, 带上最初的配置是 $B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B_0$ 。

1. 将对数字零的表示添加到输入的右侧。最小化操作符要进行的查询从下面的带子配置开始:

$$B x_1 B \bar{x}_2 B \dots B x_n B 0 B.$$

输入右侧的数字, 称为 j , 是最小化操作符的索引。

2. 制造一份参数和 j 的工作拷贝, 从而在带上产生配置

$$B x_1 B \bar{x}_2 B \dots B x_n B j B x_1 B x_2 B \dots B x_n B j B.$$

3. 利用图灵机 P 来运行包含参数和 j 拷贝的输入。在带上产生

$$B \bar{x}_1 B \bar{x}_2 B \dots B \bar{x}_n B \bar{j} B \overline{p(x_1, x_2, \dots, x_n, j)} B.$$

4. 如果 $p(x_1, x_2, \dots, x_n, j) = 1$, 那么 p 最小化的值就是 j 。否则, 将 $p(x_1, x_2, \dots, x_n, j)$ 擦除, 增加 j , 重复步骤 2 中的计算。

当出现第一个使 $p(x_1, x_2, \dots, x_n, j) = 1$ 的 j 的时候, 计算在第 4 步停机。如果这样的值不存在, 那么计算就会无限地循环下去, 表明此时 f 是没有定义的。 ■

13.7 图灵可计算函数和 μ -递归函数

我们已经证明了所有的 μ -递归函数都是图灵可计算的。现在我们把精力转换到与之相对的结论上来, 即每个图灵可计算函数都是 μ -递归函数。为了证明这一点, 我们需要设计一个数论函数来模拟图灵机的计算。为了构造这个模拟函数, 我们需要完成从图灵机的领域到自然数领域的迁移。我们把从图灵机计算到函数的转换称为图灵机的代数化 (arithmetization)。

代数化首先为图灵机的配置赋上一个值, 设 $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_n \rangle$ 是可以计算一个变量的数论函数 f 的标准图灵机。我们将构造一个 μ -递归函数来数字化地模拟 M 的计算过程。这个构造过程可以很容易地泛化到具有多个变量的函数上。

图灵机 M 的配置包含了状态、带头的位置以及从带的左边界到最右边非空白的符号之间的符号

所有的这些成分都要被表示为一个自然数,我们将用如下的方式来表示状态和带上的字母表:

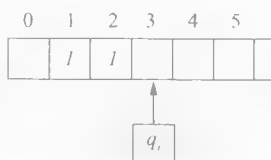
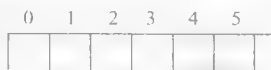
$$Q = \{q_0, q_1, \dots, q_n\}$$

$$\Gamma = \{B = a_0, I = a_1, a_2, \dots, a_k\}$$

数字将通过下标来获取,带上的符号 B 和 I 分别被表示为 0 和 1。带头的位置可以使用带上的位置来表示。

带上最右边的非空白符号结束了定义在 Σ^* 上的字符串。我们使用字母表中元素的数字表示形式来对带进行编码,使用歌德尔数字编码来对与序列 i_0, i_1, \dots, i_n 对应的字符串 $a_{i_0} a_{i_1} \dots a_{i_n}$ 进行编码。代表非空带段的数字称为带数字(tape number)。

右图中对应的机器格局中非空片段的带数字是 $2^1 3^2 5^2 = 450$ 对第三个空白位置进行显式地编码将会得到 $2^1 3^2 5^2 7^1 = 3150$, 这是另外一个表示带的带数字。在带的最右非空格的右边,任意数目的空格都可以引入到带数字上。



使用数字 0 表示空白允许我们对任意的带位置进行编码,而无需考虑在带数字中编码的带段。如果 $dec(i, z) = 0$, 而且 $pn(i)$ 能够整除 z , 那么空白就可以编码到带数字 z 中。另一方面,如果 $dec(i, z) = 0$, 而 $pn(i)$ 不能够整除 z , 那么位置 i 应该出现在带上编码段的右边。 [416] 因为带数字编码了整个非空白带上的段, 那么位置 i 必然是空白。

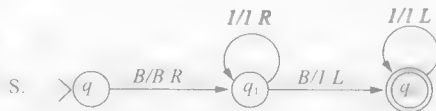
图灵机的配置是由状态数、带头位置以及带数字来确定的。配置数字将这些数字整合为一个数字。

gn_2 (状态数, 带头位置, 带数字)

这里 gn_2 是能够对有序三元组进行歌德尔数字编码的函数。

例 13.7.1 图灵机 S 完成的是后继函数。

对于计算中产生的每个配置都会给出对应的配置数字。这里需要注意带上的符号 B 和 I 分别用数字 0 和 1 来表示。



	状 态	位 置	带 数 字	配 置 数 字
$q_0 B I I B$	0	0	$2^1 3^2 5^2 = 450$	$gn_2(0, 0, 450)$
$\vdash B q_1 I I B$	1	1	$2^1 3^2 5^2 = 450$	$gn_2(1, 1, 450)$
$\vdash B I q_1 I B$	1	2	$2^1 3^2 5^2 = 450$	$gn_2(1, 2, 450)$
$\vdash B I I q_1 B$	1	3	$2^1 3^2 5^2 7^1 = 3150$	$gn_2(1, 3, 3150)$
$\vdash B I q_2 I I B$	2	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 2, 242550)$
$\vdash B q_2 I I I B$	2	1	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 1, 242550)$
$\vdash q_2 B I I I B$	2	0	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 0, 242550)$

标准图灵机中的转移不需要改变带或者状态,但是必需移动带头。带头位置的变化和歌德尔数字编码的惟一性保证了在一个计算中两个连续的配置数必然是不同的。

我们还可以构造函数 tr_M 来追踪图灵机 M 的计算过程。追踪计算的过程意味着需要产生配置数序列,这个序列对应函数计算过程中产生的配置序列。 $tr_M(x, i)$ 的值是当 M 处理输入 x 的时候,经过 i 次转换得到的配置数。因为 M 的初始配置是 $q_0 B \bar{x} B$, 于是

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

[417]

$tr_M(x, y+1)$ 的值是通过操作配置数 $tr_M(x, y)$ 从而构造接下来的机器配置而得到的。

状态和带头所指的位置上的符号决定了在图灵机 M 上要使用的转换。原始递归函数

$$cs(z) = dec(0, z)$$

$$ctp(z) = dec(1, z)$$

$$cts(z) = dec(ctp(z), dec(2, z))$$

返回状态号、带头的位置和在配置数 z 下带头所指的符号数。带头的位置是直接通过对配置数进行解码得到的。带头所指符号的数字表示被编码为带数字里面的第 $ctp(z)$ 个元素。 cs 、 ctp 和 cts 里面的 c 代表了当前配置中的部分：当前状态、当前带头位置以及当前带上的符号。

转换归约了如何对机器进行配置，也就是相应的配置号如何进行改变。在 M 中转换可以表示为

$$\delta(q_i, b) = [q_j, c, d],$$

这里的 $q_i, q_j \in Q; b, c \in \Gamma$ 并且 $d \in \{R, L\}$ 。通过定义函数来模拟 M 中转换的效果。我们首先列出了 M 中的转换：

$$\delta(q_{i_0}, b_0) = [q_{j_0}, c_0, d_0]$$

$$\delta(q_{i_1}, b_1) = [q_{j_1}, c_1, d_1]$$

$$\vdots$$

$$\delta(q_{i_m}, b_m) = [q_{j_m}, c_m, d_m].$$

图灵机的确定性保证了转换的参数都是不同的。

“新状态”的函数为

$$ns(z) = \begin{cases} j_0 & \text{如果 } cs(z) = i_0 \text{ 并且 } cts(z) = n(b_0) \\ j_1 & \text{如果 } cs(z) = i_1 \text{ 并且 } cts(z) = n(b_1) \\ \vdots & \vdots \\ j_m & \text{如果 } cs(z) = i_m \text{ 并且 } cts(z) = n(b_m) \\ cs(z) & \text{其他情况} \end{cases}$$

它返回的是在配置数 z 下执行转换所进入状态的状态号。右边的条件表示了相应的转换。设 $n(b)$ 为代表了带上符号 b 的数字；那么，第一个条件可以被解释为“如果当前状态号是 i_0 ，而当前带上的符号是 b_0 ，那么新的状态对应的数字是 j_0 ”。这里直接将原来的转换映射到数字表示上来。 M 中的每个转换都在 ns 中定义了一个条件。最后的条件表明了如果没有转换能够匹配状态和输入符号，那么新的状态和当前状态是一致的。条件定义了一组穷举的而且是相互独立的原始递归谓词。因此， $ns(z)$ 是原始递归的。计算新的带上符号数字的函数 nts 可以用完全类似的方式来定义。

计算新的带头位置的函数按照转换中的规约来改变当前的位置数。转换将方向定义为 L (左) 或者 R (右)。将带头向左移动将在当前的位置数上减一，而向右移动则是加一。为了将方向进行数字化，我们使用下面的表示

$$n(d) = \begin{cases} 0 & \text{如果 } d = L \\ 2 & \text{如果 } d = R \end{cases}$$

新的带位置函数可以定义为：

$$ntp(z) = \begin{cases} ctp(z) + n(d_0) - 1 & \text{如果 } cs(z) = i_0 \text{ 并且 } cts(z) = n(b_0) \\ ctp(z) + n(d_1) - 1 & \text{如果 } cs(z) = i_1 \text{ 并且 } cts(z) = n(b_1) \\ \vdots & \vdots \\ ctp(z) + n(d_m) - 1 & \text{如果 } cs(z) = i_m \text{ 并且 } cts(z) = n(b_m) \\ ctp(z) & \text{其他情况} \end{cases}$$

$n(d_i) - 1$ 对于当前位置号进行操作，当带头向左移动的时候，将位置数减一；当带头向右移动的时候，将位置数加一。

我们已经差不多完成了对于追踪函数的构造。给定一个机器的配置，函数 ns 和 ntp 能够计算出新配置中的状态号和带头位置。接下来要完成的就是计算新的带号。

转换将带头所指的符号进行替换。在我们函数化的方法中，带头的位置是根据配置号 z 并利用函数 ctp 得到的。将要在带上 $ctp(z)$ 位置写入的符号用数字表示为 $nts(z)$ 。新的带数字可以通过改变当前带数字中 $pn(ctp(z))$ 的指数来获得。在转换之前，对 z 进行分解可以得到 $pn(ctp(z))^{cts(z)+1}$ ，这是 $ctp(z)$ 位置上的当前符号的编码。在转换之后， $ctp(z)$ 位置上的符号表示为 $nts(z)$ 。原始递归函数

$$ntm(z) = quo(ctm(z), pn(ctp(z))^{cts(z)+1}) \cdot pn(ctp(z))^{nts(z)+1}$$

能够完成需要的替换。通过划分,我们可以从带数字 $ctn(z)$ 里面将 $ctp(z)$ 位置上符号的编码移除掉。接下来,结果又乘以 $pn(ctp(z))^{nts(z)+1}$,这样就对新的带符号进行了编码。

追踪函数 tr_M 是对原来的函数进行原始递归而定义的。这些函数模拟了在 M 上对配置执行转换的效果如同前面所示, M 的当前状态为 q_0 , 带头在位置 0。在带上从 1 开始到 $x+1$ 的位置放置的是计算的输入 x 。机器的配置被编码为

419

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

接下来的机器配置会使用前面的配置作为输入, 并使用新的状态、新的带头位置和新的带数字来获得。

$$tr_M(x, y+1) = gn_2(nts(tr_M(x, y)), ntp(tr_M(x, y)), ntm(tr_M(x, y))).$$

因为在 tr_M 中每个函数都是原始递归的, 所以我们可以总结出来 tr_M 不仅是 μ -递归的, 而且是原始递归的。但是, 追踪函数并不是我们采用的函数模拟图灵机的顶点, 它并不返回计算的结果而是返回配置数的序列。

图灵机 M 是根据数论函数 f 来进行计算的, 它对于输入 x 的结果可以利用函数 tr_M 来模拟。我们首先要注意到 M 有可能不停机, $f(x)$ 也有可能没有定义。关于停机的问题, 我们可以使用 tr_M 的结果来判定。如果 M 为配置 $tr_M(x, i+1)$ 声明了转换, 那么 $tr_M(x, i) \neq tr_M(x, i+1)$, 这是因为带头的移动改变了歌德尔数字。另一方面, 如果 M 在转换 i 之后停机, 那么 $tr_M(x, i) = tr_M(x, i+1)$, 因为当配置数反映的是停机配置的时候, 函数 nts , ntp 和 ntm 都返回前面的值。相应地, z 是第一个满足 $tr_M(x, z) = tr_M(x, z+1)$ 的数字, 机器会在 z 次转换之后停机。

因为任意图灵机在计算结束之前对于转换的次数都没有界定, 所以需要无限制的最小化来确定这个值。 μ -递归函数

$$term(x) = \mu z[eq(tr_M(x, z), tr_M(x, z+1))]$$

计算了当使用 M 来对输入 x 进行计算的时候, 在停机之前 M 执行过的转换的步数。当计算停止时, 机器的停机配置编码为 $tr_M(x, term(x))$ 。当停机时, 带上符号的形式为 $B\overline{f(x)}B$ 。停机时的带数字 tn 可以从停机的配置数

$$tn(x) = dec(2, tr_M(x, term(x))).$$

中获得。计算的结果可以通过计算带上 1 的个数, 或者通过计算在最终的带数字中 2 的幂指数个数来确定。后面的计算可以利用有界和来实现

$$sim_M(x) = \left(\sum_{i=0}^x eq(1, dec(i, tn(x))) \right) \div 1,$$

420

这里 y 是编码在终结带数字中的带上的段的长度。边界 y 是通过原始递归函数 $gdln(tn(x))$ 计算得来的 (练习 17)。因为带上包含了 $f(x)$ 的一元表示, 因此要在有界和上减一。

当 M 存在对于输入 x 的 f 的定义时, M 的计算和对 M 的模拟都会计算 $f(x)$ 。如果 $f(x)$ 没有定义, 无界最小化就无法返回值, 因此 sim_M 也是没有定义的。 sim_M 的构造过程完成了下面定理的证明。

定理 13.7.1 每个图灵可计算函数都是 μ -递归函数。

定理 13.6.4 和定理 13.7.1 在宏观和微观的计算方法之间建立了等价性。

推论 13.7.2 一个函数是图灵可计算的, 当且仅当它是 μ 递归的。

13.8 修订的丘奇—图灵论题

从函数的角度来说, 丘奇—图灵论题将函数的有效计算性和图灵可计算性联系在一起。利用定理 13.7.2, 丘奇—图灵论题可以使用 μ -递归函数来重新建立。

丘奇—图灵论题 (修订的) 一个数论函数是可计算的当且仅当它是 μ -递归函数。

同往常一样, 我们并不给出丘奇—图灵论题的证明过程。数学家和计算机科学家都能够接受这一点, 因为这个命题已经积累了相当多的支持证据。接受丘奇—图灵论题等同于接受“图灵机是最通用

的计算设备”一样。这个定理意味着任何可以有效计算的数论函数都可以利用图灵机来计算。这一点同样可以延伸到非数字性的计算。

[421]

我们首先可以看到任何数字计算机上的计算都可以解释为数字性的计算。计算机之间通常用字符串进行通讯，而之所以使用字符串是为了方便数据的输入以及更好地解释结果。输入将利用 ASCII 或者 EBCDIC 编码转换为 0,1 的字符串。在进行转换之后，输入的字符串被视为是自然数的二进制表示。随着计算的进行，就会产生其他的由 0 和 1 组成的序列，同样是二进制的自然数。输出通常会被转换回字符串数据，因为我们通常无法理解和识别采用内部表示的输出。

利用下面的例子，我们可以设计出有效的过程来完成从字符串计算到数论计算的转换过程。歌德尔数字编码可以用来完成从字符串到数字的转换。设 $\Sigma = \{a_0, a_1, \dots, a_n\}$ 是字母表，而 f 是从 Σ^* 到 Σ^* 的函数。为了从字符串得到歌德尔数字，首先要为字母表中的每个元素指定惟一的一个数。为了简单起见，我们将使用下标来定义 Σ 中的元素。字符串 $a_{i_1} a_{i_2} \dots a_{i_y}$ 的编码将由有界乘积产生：

$$pn(0)^{i_1+1} \cdot pn(1)^{i_2+1} \cdot \dots \cdot pn(n)^{i_y+1} = \prod_{j=0}^n pn(j)^{i_j+1},$$

其中， y 是要进行编码的字符串的长度。

解码函数从歌德尔数字的素数分解中提取出每个素数成分的指数。可以利用解码函数和字母表中的数字来重新构造字符串。如果 x 是 Σ 中字符串 $a_{i_1} a_{i_2} \dots a_{i_y}$ 的编码，那么 $dec(j, x) = i_j$ 。最初的字符串可以通过将解码的结果连接在一起得到。一旦字母表中的元素都被赋予了自然数，那么编码和解码的函数就是原始递归的，因此函数是图灵可计算的。

将字符串函数 f 转换成数字的函数可以利用对字符串进行数字编码和从数字到字符串的解码来实现。

在丘奇—图灵论题的帮助下，我们可以证明函数 f 是算法可计算的，当且仅当它相应的数字函数 f' 是图灵可计算的。我们应该注意到当 f 是图灵可计算的时候，存在一个有效的过程能够获得 f 的值。计算 f 的算法包含三个步骤：

- i) 将输入字符串 u 编码为数字 x ，
- ii) 计算 $f'(x)$ ，
- iii) 对 $f'(x)$ 进行解码从而得到 $f(u)$ ，

每一步都可以使用图灵机来完成。

[422]

现在假设存在有效的过程能够计算 f 。利用编码和解码函数的可逆性，我们可以设计出一个可以有效计算 f' 的过程。 $f'(x)$ 的值可以利用如下的步骤得到：将输入 x 转换为字符串 u ，计算 $f(u)$ ，将 $f(u)$ 进行转换从而得到 $f'(x)$ 。因为存在有效的步骤能够计算 f' ，我们可以根据丘奇—图灵论题总结出 f' 是图灵可计算的。

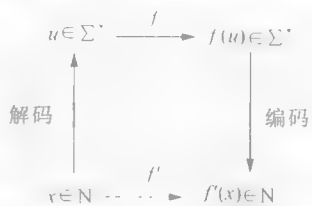
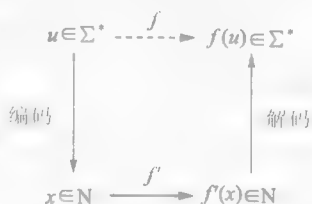
前面的证明表明了丘奇—图灵论题的影响。图灵机的计算具有普适性，而不仅限于数字计算和判定问题。一个字符串函数是可计算的，仅当该函数可以利用图灵机以及图灵可计算的编码和解码机制来实现。下面的例 13.8.1 展示了字符串函数和数字函数之间的关联关系。

例 13.8.1 设 $\Sigma = \{a, b\}$ 是字母表。下面考察交换输入字符串中 a 和 b 的函数 $f: \Sigma^* \rightarrow \Sigma^*$ 。数论函数 f' 是在计算 f 的基础上，增加了对 Σ 上字符串的编码和解码而得的。字母表中的成分使用数字 n 来进行数字化： $n(a) = 0, n(b) = 1$ 。字符串 $u = u_0 u_1 \dots u_n$ 可以编码为数字

$$pn(0)^{n(u_0)+1} \cdot pn(1)^{n(u_1)+1} \cdot \dots \cdot pn(n)^{n(u_n)+1}$$

$pn(i)$ 上的幂指数被编码 0 或者 1，而这取决于在字符串中第 i 个元素是 a 还是 b 。

设 x 是字符串 u 在 Σ 上的编码。而前面的 $gdl_n(x)$ 可以返回编码 x 所得序列的长度。因此，有界乘积



$$f'(x) = \prod_{i=0}^{gdn(x)} (eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i) + eq(dec(i, x), 1) \cdot pn(i))$$

可以产生和字符串 u 具有同样长度的编码串。如果 $eq(dec(i, x), 0) = 1$, 那么 u 中的第 i 个元素就是 a 。这在 u 的编码中由 $pn(i)^1$ 来体现。乘积

$$eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i)$$

提供了 $f'(x)$ 的因子 $pn(i)^2$ 。类似地, 在 $f'(x)$ 中, 当 u 中的第 i 个元素是 b 时, $pn(i)$ 的指数就是 1。因此, f' 构造了一个数, 这个数的素数分解可以通过将 x 中指数 1 和 2 进行交换而得到 $f'(x)$ 到字符串的转换得到了 $f(u)$ 。□

423

13.9 练习

- 设 $g(x) = x^2$, $h(x, y, z) = x + y + z$, $f(x, y)$ 是由 g 和 f 通过原始递归得到的函数。计算 $f(1, 0)$ 、 $f(1, 1)$ 、 $f(1, 2)$ 、 $f(5, 0)$ 、 $f(5, 1)$ 和 $f(5, 2)$ 的结果值。
- 仅使用基本函数、组合和原始递归操作来证明下面的函数是原始递归的。当使用原始递归的时候, 请指明函数 g 和 h 。
 - $c_2^{(3)}$
 - $pred$
 - $f(x) = 2x + 2$
- 下列的函数都是利用表 13-1 中的原始递归来定义的。请明确地给出定义中所使用的函数 g 和 h 。
 - sg
 - sub
 - exp
- 证明下列命题:
 - 由全函数 h 和 g_1, \dots, g_n 组合定义得到的函数 f 也是全函数。
 - 由全函数 h 和 g 利用原始递归得到的函数 f 也是全函数。
 - 总结得出所有的原始递归函数都是全函数。
- 设 $g = id$, $h = p_1^{(3)} + p_3^{(3)}$, f 是由函数 g 和 h 利用原始递归得到的。
 - 计算 $f(3, 0)$, $f(3, 1)$ 和 $f(3, 2)$ 的值。
 - 给出函数 f 的完整定义 (非递归的)。
- 设 $g(x, y, z)$ 是原始递归函数, 证明下面的函数都是原始递归的。
 - $f(x, y) = g(x, y, x)$
 - $f(x, y, z, w) = g(x, y, x)$
 - $f(x) = g(1, 2, x)$
- 设函数 f 为

$$f(x) = \begin{cases} x & \text{如果 } x > 2 \\ 0 & \text{其他情况} \end{cases}$$

- 给出图灵机执行 f 的状态图。
 - 证明 f 是原始递归的。
8. 证明下面的函数是原始递归的。读者可以使用表 13-1 和表 13-2 中的函数和谓词, 不允许使用有界操作符。

424

- $max(x, y) = \begin{cases} x & \text{如果 } x > y \\ y & \text{其他情况} \end{cases}$
- $min(x, y) = \begin{cases} x & \text{如果 } x < y \\ y & \text{其他情况} \end{cases}$
- $min_3(x, y, z) = \begin{cases} x & \text{如果 } x \leq y \text{ 并且 } x \leq z \\ y & \text{如果 } y \leq x \text{ 并且 } y \leq z \\ z & \text{如果 } z \leq x \text{ 并且 } z \leq y \end{cases}$

$$d) \text{ even}(x) = \begin{cases} 1 & \text{如果 } x \text{ 是偶数} \\ 0 & \text{其他情况} \end{cases}$$

$$e) \text{ half}(x) = \text{div}(x, 2)$$

$$^* f) \text{ sqrt}(x) = \lfloor \sqrt{x} \rfloor$$

9. 证明下面的谓词都是原始递归的。可以使用表 13-1 和表 13-2 以及练习 8 中的函数和谓词。不允许使用有界操作符。

$$a) \text{ le}(x, y) = \begin{cases} 1 & \text{如果 } x \leq y \\ 0 & \text{其他情况} \end{cases}$$

$$b) \text{ ge}(x, y) = \begin{cases} 1 & \text{如果 } x \geq y \\ 0 & \text{其他情况} \end{cases}$$

$$c) \text{ btw}(x, y, z) = \begin{cases} 1 & \text{如果 } y < x < z \\ 0 & \text{其他情况} \end{cases}$$

$$d) \text{ prsq}(x) = \begin{cases} 1 & \text{如果 } x \text{ 的平方根是整数} \\ 0 & \text{其他情况} \end{cases}$$

10. 设 t 是有两个变量的原始递归函数，并且定义 f 如下

$$f(x, 0) = t(x, 0)$$

$$f(x, y+1) = f(x, y) + t(x, y+1)$$

请给出借助于原始递归定义 f 的函数 g 和 h 。

11. 设 g 和 h 是原始递归函数。使用有界操作来证明下面的函数是原始递归的。这里读者可以使用任何已经证明为原始递归的谓词和函数。

425

$$a) f(x, y) = \begin{cases} 1 & \text{如果对于所有的 } 0 \leq i \leq y, \text{ 有 } g(i) < g(x) \\ 0 & \text{其他情况} \end{cases}$$

$$b) f(x, y) = \begin{cases} 1 & \text{如果对于有些 } 0 \leq i \leq y, \text{ 有 } g(i) = x \\ 0 & \text{其他情况} \end{cases}$$

$$c) f(y) = \begin{cases} 1 & \text{如果对于有些 } 0 \leq i, j \leq y, \text{ 有 } g(i) = h(j) \\ 0 & \text{其他情况} \end{cases}$$

$$d) f(y) = \begin{cases} 1 & \text{如果对于所有的 } 0 \leq i \leq y, \text{ 有 } g(i) < g(i+1) \\ 0 & \text{其他情况} \end{cases}$$

$$e) \text{ nt}(x, y) = \text{在 } 0 \leq i \leq y \text{ 范围内, } g(i) = x \text{ 的次数}$$

$$f) \text{ thrd}(x, y) = \begin{cases} 0 & \text{对于 } 0 \leq i \leq y, g(i) = x \text{ 不会超过三次} \\ j & \text{对于 } 0 \leq i \leq y, j \text{ 是满足 } g(i) = x \text{ 的第三个值} \end{cases}$$

$$g) \text{ lrg}(x, y) = \text{在 } 0 \leq i \leq y \text{ 范围内, 满足 } g(i) = x \text{ 的最大数}$$

12. 证明下面的函数都是原始递归的:

$$a) \text{ gcd}(x, y) = x \text{ 和 } y \text{ 的最大公约数}$$

$$b) \text{ lcm}(x, y) = x \text{ 和 } y \text{ 最小的公倍数}$$

$$c) \text{ pw2}(x) = \begin{cases} 1 & \text{如果对于某个 } n, \text{ 有 } x = 2^n \\ 0 & \text{其他情况} \end{cases}$$

$$d) \text{ twopr}(x) = \begin{cases} 1 & \text{如果 } x \text{ 是两个素数的乘积} \\ 0 & \text{其他情况} \end{cases}$$

13. 设 g 是单个变量的原始递归函数。证明下面的函数

$$f(x) = \min_{i=0}^x (g(i)) \\ = \min \{ g(0), \dots, g(x) \}$$

是原始递归的。

14. 证明当 p 和 u 为原始递归时，函数

$$f(x_1, \dots, x_n) = \mu z [p(x_1, \dots, x_n, z)]$$

是原始递归的。

15. 计算下面序列的歌德尔数字:

- a) 3,0
- b) 0,0,1
- c) 1,0,1,2
- d) 0,1,1,2,0

426

16. 确定下列歌德尔数字所编码的序列:

- a) 18,000
- b) 131,072
- c) 2,286,900
- d) 510,510

17. 证明下列函数是原始递归的:

- a) $gdn(x) = \begin{cases} 1 & \text{如果 } x \text{ 是某个序列的歌德尔数字} \\ 0 & \text{其他情况} \end{cases}$
- b) $gdln(x) = \begin{cases} n & \text{如果 } x \text{ 是某个长度为 } n \text{ 序列的歌德尔数字} \\ 0 & \text{其他情况} \end{cases}$
- c) $g(x,y) = \begin{cases} 1 & \text{如果 } x \text{ 是歌德尔数字, 而 } y \text{ 在 } x \text{ 的编码序列中出现} \\ 0 & \text{其他情况} \end{cases}$

18. 构造一个原始递归函数, 它的输入是编码的有序对, 它的输出也是有序对, 但是有序对中元素的位置进行了交换。例如, 如果输入是 $[x,y]$, 那么输出就是 $[y,x]$ 。

19. 函数 f 定义如下:

$$f(x) = \begin{cases} 1 & \text{如果 } x=0 \\ 2 & \text{如果 } x=1 \\ 3 & \text{如果 } x=2 \\ f(x-3) + f(x-1) & \text{其他情况} \end{cases}$$

请给出 $f(4)$ 、 $f(5)$ 和 $f(6)$ 的值。证明 f 是原始递归的。

*20. 设 g_1 和 g_2 是单个变量的原始递归函数。设 h_1 和 h_2 是四个变量的原始递归函数。两个函数 f_1 和 f_2 定义如下:

$$\begin{aligned} f_1(x,0) &= g_1(x) \\ f_2(x,0) &= g_2(x) \\ f_1(x,y+1) &= h_1(x,y,f_1(x,y),f_2(x,y)) \\ f_2(x,y+1) &= h_2(x,y,f_1(x,y),f_2(x,y)) \end{aligned}$$

这里 f_1 和 f_2 被称为从 g_1 和 g_2 以及 h_1 和 h_2 进行了同步递归 (simultaneous recursion)。 $f_1(x,y+1)$ 和 $f_2(x,y+1)$ 都是使用利用前面的函数值定义的。证明 f_1 和 f_2 是原始递归的。

427

21. 函数 f 的定义如下:

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= \sum_{i=0}^y f(i)^y. \end{aligned}$$

- a) 计算 $f(1)$ 、 $f(2)$ 和 $f(3)$ 的值。
- b) 使用串值递归来证明 f 是原始递归的。

22. 设函数 A 是阿克曼函数 (见 13.6 节),

- a) 计算 $A(2,2)$ 。
- b) 证明对于每个 $x,y \in \mathbb{N}$, $A(x,y)$ 的值是惟一的。
- c) 证明 $A(1,y) = y+2$ 。
- d) 证明 $A(2,y) = 2y+3$ 。

23. 证明下面的函数都是 μ 递归的。函数 g 和 h 都是原始递归的。

$$\text{a) } \text{cube}(x) = \begin{cases} 1 & \text{如果 } x \text{ 的平方根是整数} \\ \uparrow & \text{其他情况} \end{cases}$$

b) $\text{root}(c_0, c_1, c_2) =$ 四元多项式 $c_2 \cdot x^2 + c_1 \cdot x + c_0$ 的最小自然数根

$$\text{c) } r(x) = \begin{cases} 1 & \text{如果对于某个 } i \geq 0, \text{ 有 } g(i) = g(i+x) \\ \uparrow & \text{其他情况} \end{cases}$$

$$\text{d) } l(x) = \begin{cases} \uparrow & \text{如果对于所有的 } i \geq 0, \text{ 有 } g(i) - h(i) < x \\ 0 & \text{其他情况} \end{cases}$$

$$\text{e) } f(x) = \begin{cases} 1 & \text{如果对于某个 } i, j \in \mathbb{N}, \text{ 有 } g(i) + h(j) = x \\ \uparrow & \text{其他情况} \end{cases}$$

$$\text{f) } f(x) = \begin{cases} 1 & \text{如果对于某个 } y > x, z > x, \text{ 有 } g(y) = h(z) \\ \uparrow & \text{其他情况。} \end{cases}$$

*24. 无界的 μ 操作符可以用来定义下面的部分谓词:

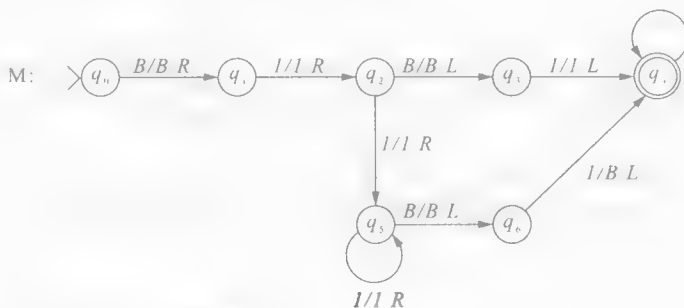
$$\mu z[p(x_1, \dots, x_n, z)] = \begin{cases} j & \text{如果对于 } 0 \leq i < j, \text{ 有 } p(x_1, \dots, x_n, i) = 0, \text{ 而且 } p(x_1, \dots, x_n, j) = 1 \\ \uparrow & \text{其他情况。} \end{cases}$$

即, 在出现第一个满足 $p(x_1, \dots, x_n, j) = 1$ 的 j 之前, 对于某些 i , 如果 $p(x_1, \dots, x_n, j) \uparrow$ 那么值是没有定义的。证明通过将定义 13.6.3 中的无界最小化操作符替换为前面的 μ 操作符, 可以得到的函数家族是图灵可计算函数。

428

25. 为例 13.7.1 里面的图灵机 S 构造 ns 、 ntp 和 nts 函数。

26. 设 M 如下



a) M 计算的是什么样的一元数论函数?

b) 在 M 对于输入为 $\bar{0}$ 的执行计算的过程中, 给出每个配置的带数字。

c) 在 M 对于输入为 $\bar{2}$ 的执行计算的过程中, 给出每个配置的带数字。

27. 假设函数 f 定义如下:

$$f(x) = \begin{cases} x+1 & \text{如果 } x \text{ 是偶数} \\ x-1 & \text{其他情况。} \end{cases}$$

a) 给出计算 f 的图灵机 M 的状态图。

b) 追踪机器对于输入 1 ($B11B$) 的计算过程。给出计算过程中每个配置对应的带数字。对于计算中的每一步, 给出 $tr_M(1, i)$ 的值。

c) 证明 f 是原始递归的。你可以使用在 13.1 节、13.2 节和 13.4 节中已经证明是原始递归的函数。

*28. 设 M 为图灵机, 而 tr_M 是 M 的追踪函数

a) 证明函数

$$\text{prt}(x, y) = \begin{cases} 1 & \text{如果 } M \text{ 对于输入 } x \text{ 的第 } y \text{ 次转移打印了空格} \\ 0 & \text{其他情况} \end{cases}$$

是原始递归的。

b) 证明函数

$$lprt(x) = \begin{cases} 1 & \text{如果 } M \text{ 对于输入 } x \text{ 的最后一次转移打印了 } 1 \\ \uparrow & \text{其他情况} \end{cases}$$

是 μ 递归的。

c) 借助于打印问题的不可判定性 (练习 12.7), 解释为什么 $lprt$ 不是原始递归的。

29. 给出一个不是 μ 递归函数的例子。提示: 考虑一个不是递归可枚举的语言。

30. 设 f 为从 $\langle a, b \rangle$ 到 $\langle a, b \rangle$ 的函数, 定义为 $f(u) = u^R$ 。构造原始递归函数 f' , 使得它能够结合编码函数和解码函数计算 f 。

31. 如果一个数论函数可以通过分别执行前驱函数和后继函数的图灵机 S 和 D 以及在 9.3 节定义的宏来计算; 那么这个函数就可以被称为宏可计算的 (macro-computable)。证明每个 μ -递归函数都是宏可计算的。为此, 需要证明:

i) 后继函数、零函数和投影函数都是宏可计算的。

ii) 宏可计算函数在组合、原始递归和无界最小化操作下面是闭合的。

32. 证明在 9.6 节中定义的编程语言 TM 能够计算整个 μ -递归函数的结合。

参考文献注释

关于函数和机械的可计算性的研究在上世纪 30 年代得到了很大的发展。Gödel [1931] 定义了一种计算的方法, 现在被称为 Herbrand-Gödel 可计算性。Kleene 对 Herbrand-Gödel 可计算性和 μ -递归函数的性质进行了深入的研究。 μ -递归函数和图灵可计算函数的等价性由 Kleene [1936] 证明。波斯特系统 [Post, 1936] 提供了另外一种执行数字计算的方法。Kleene [1952] 的经典著作提出了可计算性、丘奇-图灵论题和递归函数。关于递归函数理论的进一步研究可以在 Hermes [1965]、Péter [1967] 和 Rogers [1967] 中找到。Hennie [1977] 在抽象的算法家族概念中提出了可计算性。

阿克曼函数是由 Ackermann [1928] 提出的。对于阿克曼函数性质的分析可以在 Hennie [1977] 中找到。

第四部分

计算复杂性

前面各章的主要目的是刻画可解问题和可计算函数的集合。现在我们开始把注意力从证明问题存在算法解转向分析它们的复杂性。复杂性是由确定解所需要的资源来度量的。因此，正如引言中首先提出的，我们开始形式化地分析多少的问题。

复杂性理论致力于区分在实践中可解的问题和那些仅仅在原理上是可解的问题。理论上可解的问题并不一定有实际的解；也许不存在能够解决这个问题、同时又不需要特别大的时间或存储资源的算法。不存在有效算法的问题被称为难解问题（intractable）。

既然我们感兴趣的是问题内在的复杂性，因此分析应该独立于任何特定的实现。为了将问题的特征从实现特征中分离出来，必须选择一个单独的算法系统用于分析计算复杂性。同时这个选择不应该在计算上附加任何不必要的约束，诸如时间或可用存储的限制，因为这些限制是实现的属性而不是算法本身的属性。标准图灵机满足以上所有要求，并能为分析问题复杂性提供基础的计算框架。此外，丘奇-图灵论题确保任何有效的过程都能在这样的机器上实现。

图灵机的时间和空间复杂性分别度量转换的数量和计算中所需的带的数量。使用确定型图灵机在多项式时间内可解的 \mathcal{P} 类问题通常被认为包含了所有有效的可解问题。另一类问题， \mathcal{NP} 问题，包含了所有使用非确定型图灵机在多项式时间内可解的判定问题。显然， \mathcal{P} 是 \mathcal{NP} 的一个子集。当前，这两类问题是否是等价的仍然无从得知。

使用非确定型解的猜想。检查策略（guess-and-check strategy）， \mathcal{NP} 类包含所有在多项式时间内可对解进行验证的问题。回答 $\mathcal{P} = \mathcal{NP}$ 问题等价于判定针对一个问题构造解在本质上是否比检查一个可能性是否是解更困难。虽然这似乎是理所当然的，然而尚未被形式化地证明。

一个问题是 \mathcal{NP} 完全的，如果 \mathcal{NP} 类中的每个问题都可在多项式时间内被化简为该问题。那么为一个 \mathcal{NP} 完全问题找到多项式时间解就是足以证明 $\mathcal{P} = \mathcal{NP}$ 。但是，到目前为止仍然没有发现这样的算法。此外，大多数计算机科学家和数学家并不相信存在这样的算法。对 \mathcal{NP} 完全性的检查开始于通过显式地将 \mathcal{NP} 类中的问题化简为 \mathcal{NP} 完全问题，从而说明可满足性问题是 \mathcal{NP} 完全的。

许多学科的问题都被证明是 \mathcal{NP} 完全的，这些学科包括模式识别、调度、决策分析、组合数学、网络设计和图论等。判定一个问题是 \mathcal{NP} 完全的并不意味着不再需要解，而仅仅表明很可能不存在能在多项式时间内求解的算法。人们常常使用近似算法高效地为 \mathcal{NP} 完全的最优化问题生成近似最优解。为了说明在获取近似解时常采纳的策略，我们将介绍几个著名 \mathcal{NP} 完全问题的近似求解算法，这些算法能够在预定义的精度范围内生成近似解。

第 14 章 时间复杂性

我们从确定型图灵机的时间复杂性分析研究计算复杂性。确定型图灵机的时间是由计算中转换的数量度量的。由于用同样长度字符串初始化的计算中的转换数量不同，因此通常使用增长的速度来描述时间复杂性。我们将会看到实现于确定型多道和多带图灵机的算法的时间复杂性与实现于标准图灵机的算法的时间复杂性仅仅有多项式的差别。

语言的时间复杂性是由接收该语言的机器决定的。这里将介绍语言复杂性的若干重要属性。首先，在时间复杂性方面，没有接收一种语言的最好的图灵机。任何需要的线性因素（linear factor）都可用来为一个接收语言的机器“加速”，从而生成另外一个复杂性小的机器。加速原理（speedup theorem）可以生成一个更快的机器，但是其时间复杂性与原机器有同样的增长速度。同时，我们还将介绍这样一种语言，对其而言不存在有最小渐近时间复杂度的图灵机。对于任何接收这种语言的图灵机，我们都能构造另外一个（严格来说）增长速率更小，从而时间复杂性更小的图灵机。最后，本章将阐述不存在语言时间复杂性的上界。对任何的计算函数而言，都存在一种语言，其复杂性不会被该函数的值所限定。

[433]

丘奇-图灵论题保证了任何一个使用现代计算机技术可解的问题对于图灵机都是可解的，但这却与两个系统计算的复杂性无关。本章我们将介绍使用图灵机模拟计算机的计算；同时，图灵机中转换的数量仅仅随着计算机执行指令的数量多项式地增长。因此，图灵机必需的资源边界能够提供关于算法复杂性和计算机程序的有用信息。

14.1 复杂性度量

计算复杂性研究的两个主题是解决一个特定问题的算法的评估和不同问题固有难度的比较。本部分内容要关注的是后一个问题，但是问题复杂性的比较要求我们具有分析解决实际问题的算法的能力。为了了解算法分析的主要问题，我们首先考虑度量以下四个相似问题的时间复杂性：

为一个整数数组排序

输入：数组 $A[1..n]$

输出：排序的数组 $A'[1..n]$

求矩阵的平方

输入：整数矩阵 $B_{n \times n}$

输出：矩阵 $C = B^2$

有向图的路径问题

输入：图 $G = (N, A)$ ，节点 $v_i, v_j \in N$

输出：是；如果 G 中存在一条从 v_i 到 v_j 的路径

否；否则

图灵机 M （对任何输入都停机）的接收性（acceptance）

输入：字符串 w

输出：是；如果 M 接收 w

否；否则

前两个问题的计算中所描述的算法计算函数。排序问题在数组之间进行映射，求平方问题在矩阵

之间进行映射。路径和接收性问题是判定问题，其结果为“是”或者“否”。

复杂性函数描述了解决一个问题所需要的资源或者步骤的数目，被度量的内容随着问题的不同而有所变化：数据移动的数量、执行的算术操作的数量、执行的指令的数量、使用的空间的数量等等。其目标不是计算每个可能输入的精确的资源需求，而是提供信息以确保对每一个输入而言可以获得充足的资源。 [434]

一个算法的复杂性分析需要一下三項内容：识别出需要考虑的资源，根据输入的复杂性对输入实例分类，构造一个能将输入复杂性和资源利用关联起来的函数。

识别出需要度量的资源后，下一步工作就是要将输入实例的集合分类。分类中的每一个集合包含了具有相似属性的实例，并且对于任何一个这样的集合，其包含的实例的复杂性都可以用一个相关的自然数来表示。表14-1给出了前面所述四个例子问题的输入域的标准分类。例如，排序问题的输入实例按照数组的大小分组，分类的结果是一系列的集合 A_0, A_1, A_2, \dots ，其中 A_i 包含所有大小为 i 的数组。数字 i 是与集合 A_i 的实例相关的输入复杂性。

表 14-1 复杂性函数的组成

问题	输入复杂性	需要度量的资源占用状况
排序问题	数组的大小	数据移动的数量
求平方问题	矩阵的维度	数值乘法数量
路径问题	图中节点的数目	搜索中访问的节点数量
接收性问题	输入字符串的长度	计算中转换的数量

用 P 表示一个输入实例被分类为复杂性类 I_0, I_1, I_2, \dots 的问题，其中下标表示每个类的数字化的复杂性。 P 的一个解的复杂性函数描述一个类 I_i 中任何问题实例的最大资源占用。也就是说，一个复杂性函数是从自然数（输入的复杂性度量）到自然数（资源使用）的映射，这个映射表示了类 I_i 中每一个问题实例资源占用的上界。

当我们比较解决同样的问题的算法时，输入的复杂性和需要考虑的资源通常会在特定于问题的内容中给出。例如，分析排序问题时使用的度量与表14-1类似。冒泡排序，归并排序和插入排序的输入都是数组并且这些排序算法都移动数据。因此，根据数据移动的次数来比较这些算法的效率就是合理的。

当比较解决不同问题的算法时，特定于问题的度量就毫无意义了。一个排序算法的数据移动次数与一个图遍历时访问的节点数量之间有什么关系？即使我们知道每个算法的复杂性函数，我们也无法比较算法的效率或者问题的相对难度。输入复杂性的估计更加不合理，我们没有理由相信为一个大小为 n 的数组排序所需要的资源应该与搜索一个包含 n 个节点的图所需要的资源相关。然而，以这些问题的高层组件的形式而定义的复杂性函数会给出这样的信息。 [435]

为了比较不同的问题，解决方案必须在一个通用的算法系统中实施，这样才能通过同样的输入度量和资源使用来分析复杂性。图灵机为问题复杂性的研究提供了理想的算法系统。对一个计算而言，图灵机没有人为的存储限制和可用时间限制。此外，丘奇—图灵论题为任何有效的过程都能在图灵机上应用提供了保证。所有问题共同的输入度量是输入字符串的长度。图灵机的时间和空间复杂性描述了一个计算所需的转换的次数和带上方格的数目。

14.2 增长的速度

有时候获取输入复杂性和资源使用之间的精确的关系相当困难，并且它们提供的信息几乎总是多于我们需要的信息。正因如此，时间复杂性通常使用复杂性函数的增长速度来表示而不是使用复杂性函数本身。在继续评估算法复杂性之前，我们要首先回顾一下函数增长速度的数学分析。

一个函数的增长速度对输入任意变大时函数的渐近的性能进行度量。直觉上，增长速度取决于对函数增长影响最大的项。表14-2通过检查函数 n^2 和函数 $n^2 + 2n + 5$ 的增长来度量独立的项对一个函数的影响。我们通过最下面一行 n^2 和 $n^2 + 2n + 5$ 的函数的比值来度量它们在增长中所起到的作用。函

数 $n^2 + 2n + 5$ 的线性项和常量项称为低阶项 (low-order item)。低阶项可以对函数的初始值有不正当的影响。当 n 变大时, 低阶项显然不能显著影响函数值的增长。我们将介绍使用函数的阶和“大 O”表示法来描述函数值的渐近增长。

表 14-2 函数的增长

n	0	5	10	25	50	100	1,000
n^2	0	25	100	625	2,500	10,000	1,000,000
$n^2 + 2n + 5$	5	40	125	680	2,605	10,205	1,002,005
$n^2 / (n^2 + 2n + 5)$	0	0.625	0.800	0.919	0.960	0.980	0.998

定义 14.2.1 设函数 $f: \mathbb{N} \rightarrow \mathbb{N}$ 和函数 $g: \mathbb{N} \rightarrow \mathbb{N}$ 是单变量的数论函数。

i) 函数 f 称为 g 的阶, 若存在正常数 c 和自然数 n_0 , 使得对一切 $n \geq n_0$ 有 $f(n) \leq c \cdot g(n)$ 。

ii) g 的阶的所有函数的集合表示为 $O(g) = \{f \mid f \text{ 是 } g \text{ 的阶}\}$, 叫做“ g 的大欧”。

[436]

当一个函数 f 的值在 g 值的常数倍范围内时, f 是 g 的阶。由于低阶项的影响, $f(n) \leq c \cdot g(n)$ 中的不等关系仅仅用于控制那些比特定数字大的输入。当 f 是 g 的阶时, 我们可以说 g 为 f 提供了一个渐近的上界。

在传统意义上, $f = O(g)$ 用于表示 f 是 g 的阶。因为 $O(g)$ 是一个集合, 所以更准确的写法是 $f \in O(g)$ 。这里使用不是那么传统的“=”来表示 f 是 g 的阶是因为, 通常在一些表达式中 $O(g)$ 表示的是集合中的任意一个元素。例如, 一个函数可以写作 $f(n) = n^2 + O(n)$, 这表示 f 由一个 n^2 加上某些低阶的并且近似边界是 n 的项组成, 而不是某些特定的低阶项。我们将用 $f \in O(g)$ 表示 f 是 g 的阶, 通常读作“ f 是 g 的大欧”。

例 14.2.1 设 $f(n) = n^3$, $g(n) = n^2$, 那么 $f \in O(g)$ 并且 $g \notin O(f)$ 。显然, $n^2 \in O(n^3)$, 因为 $n^2 \leq n^3$ 对所有自然数都成立。

如果我们假设 $n^3 \in O(n^2)$ 成立, 那么必然存在常数 c 和 n_0 , 使得对于所有 $n \geq n_0$ 都有

$$n^3 \leq c \cdot n^2 \quad n \geq n_0.$$

成立。选择的 $n_0 + 1$ 和 $c + 1$ 最大值 n_1 , 那么 $n_1^3 = n_0 \cdot n_1^2 > c \cdot n_1^2$ 且 $n_1 > n_0$, 与以上不等式矛盾。因此我们的假设错误, 即 $n^3 \notin O(n^2)$ 。□

如果函数 $f \in O(g)$ 并且 $g \in O(f)$, 那么我们说 f 和 g 具有同样的增长速度。当 f 和 g 具有同样的增长速度时, 根据定义 14.2.1 有以下两个不等式

$$\begin{aligned} f(n) &\leq c_1 \cdot g(n) & n \geq n_1 \\ g(n) &\leq c_2 \cdot f(n) & n \geq n_2, \end{aligned}$$

其中 c_1 和 c_2 是正常数。结合以上两个不等式我们就会看到, 每个函数的上界和下界都是另外一个函数与常数的乘积:

$$\begin{aligned} f(n)/c_1 &\leq g(n) \leq c_2 \cdot f(n) \\ g(n)/c_2 &\leq f(n) \leq c_1 \cdot g(n). \end{aligned}$$

[437]

这些关系对于所有比 n_1 和 n_2 大的 n 都成立。因为这些边界的约束, 所以很显然 f 和 g 都不可能比对方增长的快。

例 14.2.2 设 $f(n) = n^2 + 2n + 5$ 且 $g(n) = n^2$, 那么 $f \in O(g)$ 并且 $g \in O(f)$ 。因为

$$n^2 \leq n^2 + 2n + 5$$

对所有自然数成立, 即设 c 为 1 并且 n_0 为 0 时上式满足定义 14.2.1 的条件, 所以 $g \in O(f)$ 。

为了证明反方向的关系, 我们首先说明当 $n \geq 1$ 时 $2n \leq 2n^2$ 且 $5 \leq 5n^2$, 所以有

$$\begin{aligned} f(n) &= n^2 + 2n + 5 \\ &\leq n^2 + 2n^2 + 5n^2 \\ &= 8n^2 \\ &= 8 \cdot g(n) \end{aligned}$$

对 $n \geq 1$ 成立。在大 O 的术语中, 以上不等式即表示 $n^2 + 2n + 5 \in O(n^2)$ 。□

如果 f 和 g 具有同样的增长速度, 那么 g 叫做 f 的渐进紧密边界 (asymptotically tight bound) 集合

$$\Theta(g) = \{f \mid f \in O(g) \text{ 且 } g \in O(f)\}$$

包含了所有以 g 为渐进紧密边界的函数。在大 O 表示法中, 我们用 $f \in \Theta(g)$ 表示 g 是 f 的渐进紧密边界。

一个整系数多项式是一个如下所示的函数

$$f(n) = c_r \cdot n^r + c_{r-1} \cdot n^{r-1} + \cdots + c_1 \cdot n + c_0,$$

其中 c_0, c_1, \dots, c_r 是任意整数, c_r 是非零整数, r 是正整数。常量 c_r 是函数 f 的系数, r 是多项式的次数。负数系数会导致函数产生负值。例如, 如果 $f(n) = n^2 - 3n - 4$, 那么 $f(0) = -4, f(1) = -6, f(2) = -6, f(3) = -4$ 。对所有的自然数, 多项式 $g(n) = -n^2 - 1$ 永远小于零。

以上我们在数论函数上定义了增长的速度。绝对值函数可用于将任意值多项式转换为数论函数。一个整数 i 的绝对值是一个非负整数, 其定义如下:

$$|i| = \begin{cases} i & \text{如果 } i \geq 0 \\ -i & \text{否则} \end{cases}$$

438 |

使用绝对值组合 f 将产生一个数论函数 $|f|$ 。多项式 f 的增长速度定义为 $|f|$ 的增长速度。

例 14.2.1 和例 14.2.2 介绍的技术可用于证明多项式的次数与其增长速度之间的一般关系。

定理 14.2.2 设 f 是一个 r 次多项式, 那么

- i) $f \in \Theta(n^r)$,
- ii) $f \in O(n^k)$ 对所有 $k > r$ 成立,
- iii) $f \notin O(n^k)$ 对所有 $k < r$ 成立。

定理 14.2.2 的一个结论是任何一个多项式的增长率可由一个 n^r 形式的函数刻画。第一个式子表示 r 次多项式与 n^r 有相同的增长速度。此外, 式子 (ii) 和式子 (iii) 说明, 当 k 与 r 不同时, r 次多项式的增长速度与 n^k 不同。

用于度量算法性能的其他的重要函数还包括对数函数、指数函数和阶乘函数。一个以 a 为底的数论对数函数定义如下:

$$f(n) = \lfloor \log_a(n) \rfloor.$$

可以利用常数乘积改变对数函数的底数从而改变其值。具体如下:

$$\log_a(n) = \log_a(b) \log_b(n).$$

这个定义说明对数函数的增长速度与其底数无关。

例题 14.2.1 和例题 14.2.2 使用了大 O 的定义来比较多项式函数的增长速度。当函数更加复杂时, 通常使用极限来判定两个函数的渐进复杂性会更加容易一些。设 f 和 g 是两个数论函数, 那么

1. 如果 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, 那么 $f \in O(g)$ 并且 $g \notin O(f)$ 。
2. 如果 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c (0 < c < \infty)$, 那么 $f \in \Theta(g)$ 并且 $g \in \Theta(f)$ 。
3. 如果 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, 那么 $f \notin O(g)$ 并且 $g \in O(f)$ 。

用这种方式来判定函数的增长速度通常需要首先应用洛必达法则获取极限。

[439]

用于复杂性分析的洛必达法则断言, 当 n 趋于无穷时, 如果 f 和 g 是从 \mathbf{R}^+ 到 \mathbf{R}^+ 的函数并且是连续的可微的, 那么

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)},$$

此处 f' 和 g' 分别是 f 和 g 的导数。例 14.2.3 使用极限和洛必达法则证明了对于任何底数为 a 的对数函数都有 $n \log_a(n) \in O(n^2)$ 。

例 14.2.3 设 $f(n) = n \log_a(n)$ 且 $g(n) = n^2$, 应用洛必达法则推倒 $f(n)/g(n)$ 如下:

$$\begin{aligned}
\lim_{n \rightarrow \infty} \frac{n \log_a(n)}{n^2} &= \lim_{n \rightarrow \infty} \frac{\log_a(n) + n(\log_a(e)/n)}{2n} \\
&= \lim_{n \rightarrow \infty} \frac{\log_a(n)}{2n} + \lim_{n \rightarrow \infty} \frac{\log_a(e)}{2n} \\
&= \lim_{n \rightarrow \infty} \frac{\log_a(e)/n}{2} + 0 \\
&= \lim_{n \rightarrow \infty} \frac{\log_a(e)}{2n} \\
&= 0.
\end{aligned}$$

其中, e 是自然对数的底数。由于极限是 0, 所以 $f \in O(g)$ 。 □

定理 14.2.3 比较了对数函数、指数函数和阶乘函数以及多项式函数的增长, 证明留作练习

定理 14.2.3 设 r 是一个自然数, a 和 b 是大于 1 的实数, 那么

- i) $\log_a(n) \in O(n)$
- ii) $n \notin O(\log_a(n))$
- iii) $n^r \in O(b^n)$
- iv) $b^n \notin O(n^r)$
- v) $b^n \in O(n!)$
- vi) $n! \notin O(b^n)$ 。

如果函数 f 对某些自然数 r 有 $f \in O(n^r)$ 成立, 那么称 f 是有界多项式 (polynomially bounded)。尽管不是多项式, 但根据例题 14.2.3 可知 $n \log_a(n)$ 的边界是多项式 n^2 。包括多项式在内的有界多项式函数是函数的重要组成部分, 它们与有效算法的时间复杂性密切相关。式 (iv) 和式 (vi) 说明指数函数和阶乘函数不是多项式边界的。根据定理 14.2.2 和定理 14.2.3 给出的关系, 表 14-3 显示了大 O 的层次, 按照增长速度给出了一个函数列表。在标准实践中, 如果函数 f 满足 $2^n \in O(f)$, 那么我们说 f 拥有指数级增长 (exponential growth)。根据这个惯例, n^n 和 $n!$ 都是指数级增长。

表 14-3 大 O 的层次

大 O	渐近上界
$O(1)$	常数
$O(\log_a(n))$	对数
$O(n)$	线性的
$O(n \log_a(n))$	$n \log n$
$O(n^2)$	二次的
$O(n^3)$	三次的
$O(n^r)$	多项式的 ($r \geq 0$)
$O(b^n)$	指数的 ($b > 1$)
$O(n!)$	阶乘的

一个算法的效率通常由其增长速度来刻画。一个多项式算法是指它的复杂性是多项式边界的。也就是说, 对于某些 $r \in \mathbb{N}$ 有 $c(n) \in O(n^r)$ 。当考虑到输入规模变大时函数的增长速度时, 多项式算法与非多项式算法之间的区别非常明显。表 14-4 列出了一个复杂性不是多项式的算法所需要的庞大资源。

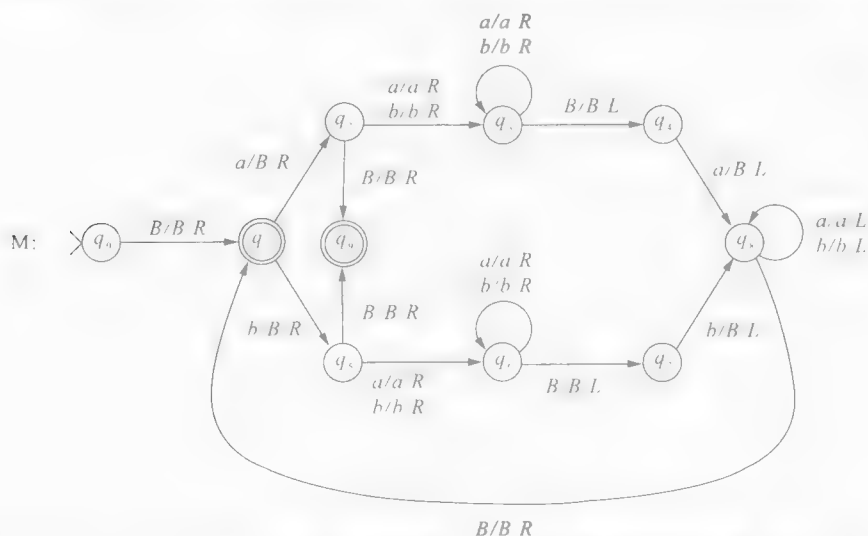
表 14-4 几个普通函数的增长

n	$\log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1,000	1,024	3,628,800
20	4	20	400	8,000	1,048,576	$2.4 \cdot 10^{18}$
30	4	30	900	27,000	$1.0 \cdot 10^9$	$2.6 \cdot 10^{32}$
40	5	40	1,600	64,000	$1.1 \cdot 10^{12}$	$8.1 \cdot 10^{47}$
50	5	50	2,500	125,000	$1.1 \cdot 10^{15}$	$3.0 \cdot 10^{64}$
100	6	100	10,000	1,000,000	$1.2 \cdot 10^{30}$	$> 10^{157}$
200	7	200	40,000	8,000,000	$1.6 \cdot 10^{60}$	$> 10^{374}$

14.3 图灵机的时间复杂性

计算的时间复杂性度量了计算要承担的工作量。图灵机计算的时间复杂性通过其处理的转移的数

量来度量。我们通过分析一个接收来自字符表 $\{a, b\}$ 的回文的机器 M 的计算来讨论确定图灵机时间复杂性要考虑的问题。



M 的计算包含一个用于比较带上的第一个和最后一个非空符号的循环。通过从 q_1 开始的一个转换，第一个符号被记录并被替换为空白。根据始于 q_1 的路径，最后一个非空的符号被检查是否与 q_1 或者 q_7 匹配。然后机器通过带上的非空片断移到左侧，同时比较循环将会重复进行。当状态 q_5 或者 q_9 读到空白时，那么该字符串是奇数长度的回文，能够在状态 q_6 被接收。偶数长度的回文在状态 q_4 被接受。

M 对符号 a 和 b 的计算是对称的。当处理字符 a 时，位于上面的那条从 q_1 到 q_8 的路径被遍历；当处理字符 b 时，下面的那条从 q_1 到 q_8 的路径被遍历。表 14-5 列出的计算包含了所有长度为 0、1、2 和 3 的字符串中字符的重要组合。

表 14-5 M 的计算

长度 0	长度 1	长度 2		长度 3	
$q_0 BB$	$q_0 BaB$	$q_0 BaaB$	$q_0 BabB$	$q_0 BabaB$	$q_0 BaabB$
$\vdash Bq_1 B$	$\vdash Bq_1 aB$	$\vdash Bq_1 aaB$	$\vdash Bq_1 abB$	$\vdash Bq_1 abaB$	$\vdash Bq_1 aabB$
	$\vdash BBq_2 B$	$\vdash BBq_2 aB$	$\vdash BBq_2 bB$	$\vdash BBq_2 baB$	$\vdash BBq_2 abB$
	$\vdash BBBq_3 B$	$\vdash BBaq_3 B$	$\vdash BBbq_3 B$	$\vdash BBbq_3 aB$	$\vdash BBaq_3 bB$
		$\vdash BBq_4 aB$	$\vdash BBq_4 bB$	$\vdash BBbaq_3 B$	$\vdash BBabq_3 B$
		$\vdash Bq_8 BBB$		$\vdash BBbq_4 aB$	$\vdash BBaq_4 bB$
		$\vdash BBq_1 BB$		$\vdash BBq_8 bBB$	
				$\vdash Bq_8 BbBB$	
				$\vdash BBq_1 bBB$	
				$\vdash BBBq_5 BB$	
				$\vdash BBBBq_9 B$	

就像我们期望的那样，这些计算显示，计算中转换的数量依赖于特定的输入字符串。实际上，处理相同长度字符串的工作量可能有本质的不同，图灵机的时间复杂性度量了处理特定长度字符串所需要的最大工作量。

定义 14.3.1 设 M 是一个标准图灵机。 M 的时间复杂性 (time complexity) 是一个函数 $tc_M: \mathbb{N} \rightarrow \mathbb{N}$ ，因此，当使用长度为 n 的字符串对图灵机 M 进行初始化时， M 处理的转换数量的最大值为 $tc_M(n)$ 。

当度量一个图灵机的时间复杂性时,我们假设图灵机的计算对每一个输入字符串都停机。试图讨论一个不确定的持续计算的效率或者更精确地弥补它的无效性是没有意义的。

定义 14.3.1 对接收语言和计算函数的机器都适用。用类似的方式可以定义确定型多道和多带图灵机。我们将会在下章讨论不确定型图灵机的复杂性。

我们对时间复杂性的定义度量了图灵机在最坏情况下的性能。我们选择用最坏情况下的性能来分析一个算法有两个原因。一个原因是我们考虑到算法计算的极限。 $tc_M(n)$ 的值描述了当使用长度为 n 的字符串对图灵机 M 进行初始化时,保证 M 能停机的最小资源。另一个原因是严格的实效,最坏情况下的性能通常比平均性能容易度量。

[443] 我们使用 M 接收 $\{a, b\}$ 上的回文的计算来说明确定时间复杂性的过程。当整个输入字符串被用空白取代或者发现第一个不匹配的字符对时, M 的计算终止。因为时间复杂性度量了最坏情况下的性能,所以我们仅仅需要关心那些计算会导致机器做最大可能的匹配。擦除循环的字符串。对 M 而言,当输入被接收时这个条件得以满足。

基于这些发现,我们可以通过表 14-5 的计算得到函数 $tc_M(n)$ 的初始值。

$$\begin{aligned} tc_M(0) &= 1 \\ tc_M(1) &= 3 \\ tc_M(2) &= 6 \\ tc_M(3) &= 10 \end{aligned}$$

要得到剩余的 tc_M 值需要对 M 的计算做更细致地分析。我们首先考虑一下处理偶数长度字符串时 M 的行为。计算在机器右移和左移的序列中转换。最初,带头处于临近左侧非空片段的位置。

- 向右移动: 带头向右移动,擦除最左端的非空符号。读入字符串的剩余部分并且机器进入状态 q_k 或者状态 q_7 。这要求 $k+1$ 次转换,此处 k 指带上非空部分的长度。
- 向左移动: M 向左移动,擦除匹配的符号,并继续通过带上的非空部分。这需要 k 次转换。

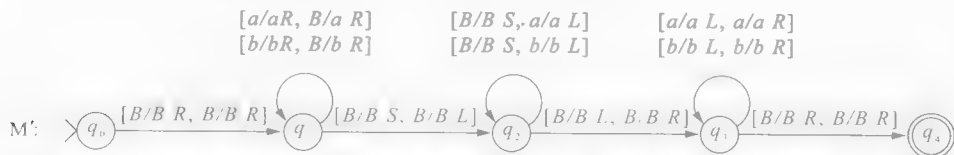
上述的活动将带上非空部分的长度减小了 2。这个比较和擦除的循环会一直重复执行直到带空为止。正如以上所述,当 M 接收输入时,偶数长度的字符串是性能最坏的情况。一个接收长度为 n 的字符串的计算需要执行上述循环 $n/2$ 次。

一个计算所需要总转换次数可以通过累计每次迭代的转换获得。正如上表所示,一个长度为偶数 n 的字符串的计算所需转换的最大次数是从 1 到 $n+1$ 的自然数之和。奇数长度字符串的分析有相同的结果。因此, M 的时间复杂性由以下函数来定义:

[444]

$$tc_M(n) = \sum_{i=1}^{n+1} i = (n+2)(n+1)/2 \in O(n^2).$$

例 14.3.1 一个双带图灵机 M'



接收字母表 $\{a, b\}$ 上的回文。 M' 的一个计算遍历输入,并将其拷贝到带 2 上,然后带 2 的头往回移动到位置 0。在这一点,带头移过输入,带 1 自右向左且带 2 自左向右地比较带 1 和带 2 上的符号。如果带头始终没有遇到不同的符号,那么输入不是一个回文,计算中止并且图灵机拒绝该输入字符串。当输入是一个回文时,计算停止并且在带 1 和带 2 同时读到空白时接收该输入的字符串。

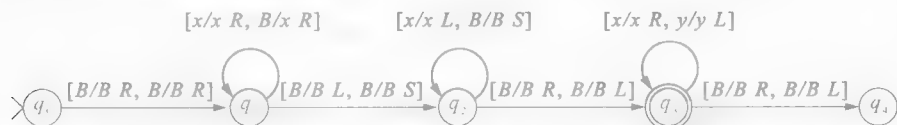
对于长度为 n 的输入,当字符串是回文时产生最大数量的转换。一个接收的计算需要三个完整的

过程：拷贝、重绕 (rewind) 和比较。通过计算每个阶段的转换数量，我们可以看到， M' 的时间复杂性是 $tc_{M'}(n) = 3(n+1) + 1$ 。

与单带图灵机相比，双带图灵机的一个转换使用了更多的信息并且实施了更为复杂的操作。正如接收回文的图灵机 M 和 M' 的复杂性所示，在转换的复杂性和必须处理的数目之间存在一个权衡。14.4 节介绍了单带图灵机和多带图灵机时间复杂性之间的精确关系。

确定一个图灵机时间复杂性的第一步是识别出能够展示最坏情况下的行为的字符串。在接收回文的图灵机中，这些就是指语言中的字符串。但是正如下面例子所示，情况也并不总是这样。

例 14.3.2 设 M 是一个双带图灵机



其中 x 和 y 可以是 a, b, c 中的任意字符，但是 $x \neq y$ 。 M 的语言包括所有这样的 a, b, c 上的字符串，它们至少包含一个值 k ，使得字符串的第 k 个字符和倒数第 k 个字符是相同的。例如， $abaa$ 、 $abccccc$ 和 $abcbbbc$ ，易知所有奇数长度的字符串都属于 $L(M)$ 。

[445]

M 的计算与例 14.3.1 中的图灵机采用相同的策略。输入的字符串被拷贝到带 2 上并且带 1 的带头返回到初始位置。随着带 1 的带头自左向右移动和带 2 的带头自右向左移动，我们可以比较带 1 和带 2 上的字符。当两个带头扫描到同样的字符时计算停止，输入字符串被接收。

对于奇数长度而言的字符串，最坏的情况是在到达中间位置之前一直没有发现匹配的字符。在这种情况下，一个长度为 n 的字符串的计算需要 $\frac{5}{2}(n+1)$ 次转换。输入字符串被 M 拒绝是偶数长度输入的性能最坏情况。在一个拒绝的计算中，带 1 的带头需要三次扫描整个输入，因此

$$tc_M(n) = \begin{cases} \frac{5}{2}(n+1) & \text{如果 } n \text{ 是奇数} \\ 3(n+1) & \text{如果 } n \text{ 是偶数} \end{cases}$$

偶数长度字符串的接收最多需要 $\frac{5}{2}(n+2)$ 次转换，这个数字永远小于最坏情况下的性能。

□

14.4 复杂性和图灵机的变种

第 8 章介绍了图灵机模型的几种变化从而来简化执行复杂计算的机器设计。在可判定性的学习中，图灵机模型的部分是不相关的。我们可以证明，任何使用某个图灵机结构可解的问题对于其他图灵机结构也都是可解的。然而，在复杂性理论中存在选择问题。14.3 节介绍的接收字母表 $\{a, b\}$ 上的回文的机器展示了单带和双带图灵机对计算资源要求的潜在不同。本节我们将讨论各种不同的图灵机模型中计算复杂性的关系。

定理 14.4.1 如果 L 是能被 k 带确定型图灵机 M 接收的语言，那么 M 的时间复杂性为 $tc_M(n)$ 。因此， L 也能够被标准图灵机 M' 接收，且时间复杂性 $tc_{M'}(n) = tc_M(n)$ 。

证明：本证明的依据是 8.4 节介绍的从 k 道图灵机构造单道图灵机的内容。单道图灵机的字母表由 M 的带字母表的符号 k 元组组成。 M 的转换的形式为 $\delta(q_i, x_1, \dots, x_k)$ ，其中 x_1, \dots, x_k 是道 1，道 2，... 道 k 上的字符。 M' 的相关转换形式为 $\delta(q_i, [x_1, \dots, x_k])$ ，其中的 k 元组是 M' 字母表中的字符。因此，对于每一个输入， M 和 M' 所处理的转换数量都是相同的，因此 $tc_M = tc_{M'}$ 。

[446]

定理 14.4.2 如果 L 是可以被 k 带确定型图灵机 M 接收的语言，那么 M 的时间复杂性 $tc_M(n) = f(n)$ 。因此， L 可以被标准图灵机 N 接受， N 的时间复杂性 $tc_N(n) \in O(f(n)^2)$ 。

证明：从 k 带图灵机构造一个等价的单带图灵机需要使用一个 $2k+1$ 道图灵机 M' 作为中介。根据定理 14.4.1，只需证明 $tc_{M'} \in O(f(n)^2)$ 即可。

本证明的依据是 8.6 节介绍的能够模拟多带图灵机行为的多道图灵机 M' 的构造。首先，我们分析

一下,为了能够模拟 M 的一次转换, M' 需要的转换的次数。

假设我们模拟 M 的第 t 次转换。最久远的情况是此时 M 的一个带头正好处于位置 t 。模拟的第一个步骤是记录奇数序号带上的被偶数带上的 X 标记的字符。这个步骤由以下几个 M' 的转移序列组成:

活 动	M' 转移的最大次数
在第二道上找到 X 并返回到带的 0 位置	$2t$
在第四道上找到 X 并返回到带的 0 位置	$2t$
\vdots	\vdots
在第 $2k$ 道上找到 X 并返回到带的 0 位置	$2t$

在找到每一个 X 下的符号以后, M' 用一个转换记录 M 的活动。因此 M 的一次转换的模拟完成了, 如下所示:

活 动	M' 转移的最大次数
在道 1 上写字符, 在道 2 上重新布置 X , 返回到带的 0 位置	$2(t+1)$
在道 3 上写字符, 在道 4 上重新布置 X , 返回到带的 0 位置	$2(t+1)$
\vdots	\vdots
在道 $2k-1$ 上写字符, 在道 $2k$ 上重新布置 X , 返回到带的 0 位置	$2(t+1)$

因此, 为了模拟 M 的第 t 次转换, M' 最多需要 $4kt + 2k + 1$ 次转换。 M' 的计算开始于一个转换, 这个转换在奇数序号的道上放置标记并且在第 $2k+1$ 道上放置 #。其余的计算则由模拟 M 的转换构成。当输入长度为 n 时, 为了模拟 M 的计算, M' 需要的转换次数的上界是:

$$tc_{M'}(n) \leq 1 + \sum_{t=1}^{f(n)} (4kt + 2k + 1) \in O(f(n)^2).$$

14.5 线性加速

图灵机 M 的时间复杂性函数 $tc_M(n)$ 给出了当输入长度为 n 的字符串时, 一次计算需要的转换次数的上界。本节我们将介绍接收语言 L 的图灵机可以被“加速”从而能够产生另外一个能够比原来快倍增常量的时间内接收 L 的机器。

设 $M = (Q, \Sigma, \Gamma, q_0, \delta, F)$ 是一个能够接收语言 L 的 $k(k > 1)$ 带图灵机。下面关于加速的策略是, 构造一个机器 N , 使其能够接收 L 并且 N 的 6 个转换能够模拟 M 的 m 个转移, 其中 m 根据需要加速的程度决定。例如, 选择 $m = 12$ 大约能够减少一半的转换次数, 因为 N 的 6 次转换可以达到与 M 的 12 次转换相同的效果。此处提到大约, 是因为 N 需要的某些初始化开销比模拟 M 的计算有优势。

因为机器 M 和机器 N 接收相同的语言, 所以 N 的输入字母表也是 Σ 。 N 的带字母表包括 M 的带字母表、符号 # 以及 Γ 的所有排序的 m 元组字符。 N 的一次计算包括两个阶段, 初始化和模拟。初始化将输入转换为一个 m 元组序列。 N 的计算的其余部分为模拟 M 的计算。

在模拟 M 的计算的过程中, N 的一个带字符是 M 字符的一个 m 元组, N 的状态用于记录那些可能被 M 的后续 m 个转移影响的 M 的带。在 N 计算的这个阶段, 其一个状态由以下几部分组成:

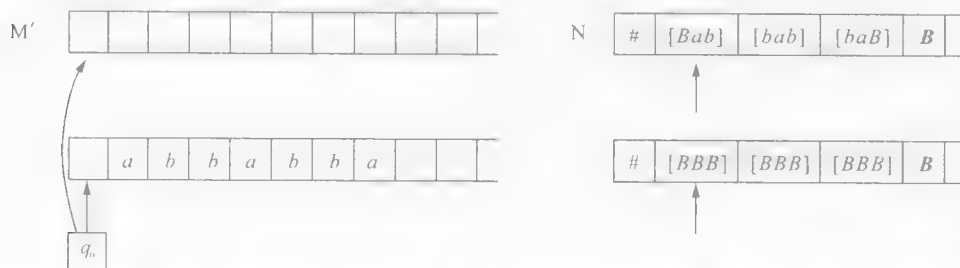
- i) M 的状态;
- ii) 从 $i=1$ 到 $i=k$, N 的带 i 上目前被扫描的 m 元组以及紧临其左右的 m 元组;
- iii) 一个排序的 k 元组 $[i_1, \dots, i_k]$, 其中, i_j 是指在正被 N 扫描的 m 元组中的带 j 上正在被 M 扫描的字符的位置。

N 的 6 个转换序列使用状态信息来模拟 M 的 m 个转移。

这个过程可以通过使用例 14.3.1 中的双带图灵机 M' 来展示, 即 $m=3$ 且输入为 *abhabba*。 N 的输入配置与 M' 相同, 其中带 1 和带 2 上的输入字符串全部为空白。 N 的第一个活动是将输入字符串编码为 m 元组。这个过程开始于在两个带的 0 位置写入字符 #。根据带 1 上每三个连续的字符, 在带 2 上写入一个排序的三元组。由于 *Babhabba* 的长度不能正好被 3 整除, 所以带 2 上有一个排序的三元组可

用空白追加 重新将N的带头布置在位置1, 并将原来的输入字符串从带1上擦除。在后续的计算中, 我们将用N的带2模拟M'的带1, 用N的带1模拟M'的带2。

下图显示了输入为 *abbabba* 时M'的初始化配置和N在编码后的配置。

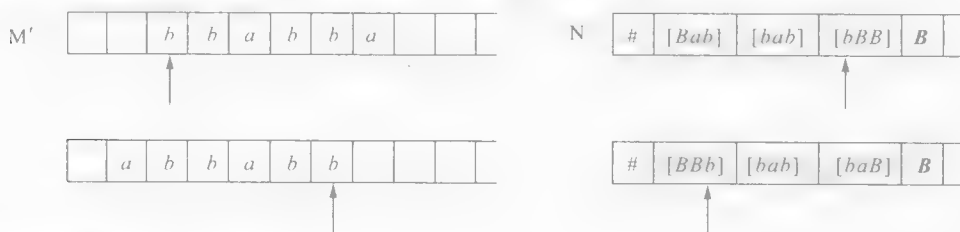


初始化后, N的带上的每一个空白将会用于放置M'的被编码的空白 [BBB]。为了在图中显示出不同, 我们把N的空白记为B。在对输入进行编码后, N将会进入以下状态:

$$(q_0; ?, [BBB], ?; ?, [Bab], ?; [1, 1]).$$

m 元组 [BBB] 和 [Bab] 是带1和带2上的当前分别正被N扫描的 m 元组。有序对 [1, 1] 显示了在N当前的状态中, M'的计算正在扫描每一个三元组 [BBB] 和 [Bab] 的第一个位置。符号? 是占位符, 后续转换将导致N进入新的状态, 这些状态中, 符号? 被与当前正被扫描的三元组向左和右的位置信息替换。

根据处理 *abbabba* 的过程中M'和N获取的配置, 下图展示了M'的 m 个动作的模拟



进入这样的配置后, N的状态是:

$$(q_3; ?, [BBb], ?; ?, [bBB], ?; [3, 1]).$$

状态中有序对 [3, 1] 说明M'的计算正在读带1上三元组 [BBb] 中的 b 和带2上三元组 [bBB] 中的 b

随后, 机器N在每条带上左移, 扫描方格, 然后进入状态

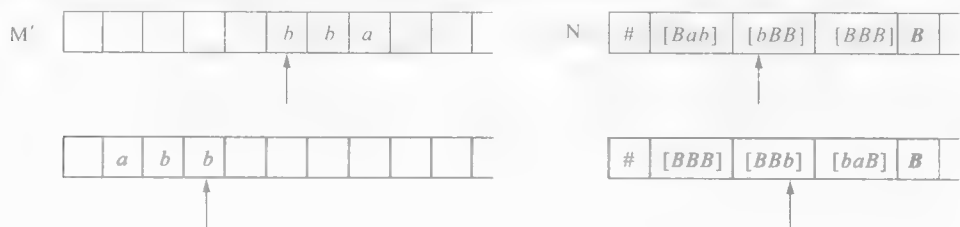
$$(q_3; \#, [BBb], ?; [bab], [bBB], ?; [3, 1]),$$

这个状态记录了原来状态中被扫描方格左边的三元组 # 的角色是保证在这个模拟阶段中N不会越过带的左侧边界。向右的两次移动会使N进入状态

$$(q_3; \#, [BBb], [bab]; [bab], [bBB], [BBB]; [3, 1]),$$

这个状态记录了原来被扫描位置的右边的三元组。随后N向左移动其带头到原来的位置。这些转换之后, N的状态包括了M'中可能被三个转移改变的带的片段的拷贝。

此时, N重新写带以便与M'在三个转移后即将进入的配置相匹配,



并且进入以下状态以便开始模拟 M' 接下来的三个转移。

$$(q_3;?, [BBb], ?;?, [bBB], ?;[3,1])$$

由于 N 的每个带方格有 M' 的三个字符, 所以 M' 能够被三个转换所改变的带包含在当前被 N 所扫描的带方格中, 并且也被包含在紧邻正在扫描的方格的左侧或者右侧的带方格中, 而不是两侧都包括。因此, N 为了更新它的带和准备继续模拟 M' 最多只需要两个转换。对 M' 的模拟会一直持续到 M' 停机, 在这种情况下 N 也会停机并且返回跟 M' 相同的组成状况。

定理 14.5.1 设 M 是接收语言 L 的 k 带图灵机 ($k > 1$), 其时间复杂性函数为 $tc_M(n) = f(n)$ 。对于任何一个常数 $c > 0$, 都存在一个 k 带图灵机 N 能够接收 L 并且 N 的时间复杂性函数 $tc_N(n) \leq \lceil cf(n) \rceil + 2n + 3$ 。

[450] 证明: 我们刚刚讨论过 N 的构造。把长度为 n 的输入字符串编码成 m 元组和重新布置带头的位置需要 $2n + 3$ 次转换。

N 剩余的計算主要是模拟 M 的計算。为了获取和记录模拟 M 的 m 次转换所需要的信息, 机器 N 执行一次左移、两次右移和一次重新将带头布置到原来位置的操作。重新配置 N 最多需要两次转换。 N 的这六个转换对于产生与 M 的 m 个转移相同的结果来说是足够了。选择 $m \geq 6/c$, 则

$$\begin{aligned} tc_N(n) &= \lceil (6/m)f(n) \rceil + 2n + 3 \\ &\leq \lceil cf(n) \rceil + 2n + 3 \end{aligned}$$

定理得证。 ■

推论 14.5.2 设 M 是能够接收语言 L 的单带图灵机, 其时间复杂性 $tc_M(n) = f(n)$ 。对于任何一个常数 $c > 0$, 都存在一个双带图灵机 N 能够接收 L 且其时间复杂性 $tc_N(n) \leq \lceil cf(n) \rceil + 2n + 3$ 。

证明: 在标准行为中, 单带图灵机 M 可以被认为是一个在计算中没有使用第二条带的双带图灵机。定理 14.5.1 可以用于加速这个双带图灵机。 ■

定理 14.5.1 中加速的代价是创建一个更大的字母表和大范围地增加状态的数量。精确地确定这些集合的大小将留作练习。

14.6 语言时间复杂性的属性

时间复杂性函数 tc_N 的定义在机器 M 而不是该机器接收的语言中显示。我们知道, 可以构造许多不同的机器接收同样的语言, 而每一个机器都可能有不同的时间复杂性。我们可以说, 如果存在一个标准的 (单带的确定型) 图灵机 M 其时间复杂性 $tc_M(n) \in O(f(n))$, 则语言 L 被在一个确定的时间 $f(n)$ 内接收。根据上一节的结果我们可以得知, 无论是否存在一个接收 L 的时间复杂性是 $O(f(n))$ 的多带图灵机, 语言 L 都是 $O(f(n)^2)$ 。

[451] 本节我们将要介绍语言时间复杂性边界的两个有趣的结果。首先, 我们将会介绍对于任何可计算全函数 $f(n)$, 都存在一个时间复杂性不在边界 $f(n)$ 内的语言。然后我们介绍存在这样的语言, 对它们而言不存在“最好”地接收它们的图灵机。定理 14.5.1 已经说明了一个接收某种语言的机器可以被线性地加速。然而, 这个处理过程不能改变该接收机器的增长速度。我们将会介绍存在这样一些语言, 它们可以被任意的机器接收, 同时也能被时间复杂性与原来的机器相比按照非常小的速率增长的机器接收。

这些结果都利用了编码和列举所有多带图灵机的能力。一个单带图灵机在对 $\{0, 1\}$ 上的字符串的编码已经在 11.5 节介绍过了。这种方法可以扩展到所有输入来自字母表 $\{0, 1\}$ 的多带图灵机编码。带字母表应该包括元素 $\{0, 1, B, x_1, \dots, x_n\}$ 。带字符按照以下原则编码:

像前面一样, 一个数字被编码为它的一元表示, 一个通过编码元素的转换由 0 分隔; 编码的转换被 00 分隔。根据这些规则, 一个 k 带图灵机可以被编码为:

字符	编码
0	1
1	11
B	111
x_1	1111
\vdots	\vdots
x_n	1^{n+3}

000 \bar{k} 000 en (接收状态)000 en (转换)000,

其中, \bar{k} 是 k 的一元表示, en 表示对圆括号内的内容编码。

根据这些表示法, 每一个字符串 $u \in \{0, 1\}^*$ 都可以被看作是某个多带图灵机的编码。如果 u 不满足编码一个多带图灵机的语法条件, 那么这个字符串被解释为一个单带单状态并且没有转换的图灵机的表示。

在练习 8.32 中构造了一个列举所有 $\{0, 1\}^*$ 上的字符串的图灵机 E 。由于每一个字符串也可以表示一个多带图灵机, 机器 E 也可以被等同地看作列举所有输入字母表为 $\{0, 1\}$ 的多带图灵机。 E 列举的字符串记做 u_0, u_1, u_2, \dots , 相应的机器记做 M_0, M_1, M_2, \dots 。

现在, 我们介绍语言的时间复杂性没有上界。更精确地说, 对于任意的可计算函数 f , 我们都能构造一个递归语言 L , 使得不存在时间复杂性 $tc_M(n) \leq f(n)$ 的图灵机 M 能够接收 L 。证明方法是, 使用对角化来获得存在这样一个图灵机的假设存在矛盾。

定理 14.6.1 设 f 是一个全函数, 那么存在一个语言 L 使得对于任何接收 L 的确定型图灵机 M , tc_M 不以 f 为边界。

证明: 设 F 是计算函数 f 的图灵机。考虑语言 $L = \{u \mid \text{在 } f(n) \text{ 或更少步骤内 } M_i \text{ 不接收 } u_i, n = \text{length}(u_i)\}$ 。首先, 我们说明 L 是递归的, 所以任何一个接收 L 的机器转移的次数都不能以 $f(n)$ 为边界。 452

一个接收 L 的机器 M 的描述如下。 M 的输入是 $\{0, 1\}^*$ 上的字符串 u , 我们再次强调字符串 u 表示的是列举所有多带图灵机的图灵机 M_i 的编码。 M 的一个计算

1. 判定 u_i 的长度, 即 $\text{length}(u_i) = n$;
2. 模拟 F 的计算以确定 $f(n)$ 。
3. 在 u_i 上模拟 M_i 直到 M_i 停机或者在此之前完成 $f(n)$ 个动作;
4. 如果 M_i 停机没有接收 u_i 或者 M_i 没能在前 $f(n)$ 个转换内停机, 那么 M 接收 u_i ; 否则拒绝 u_i 。

显然, 语言 $L(M)$ 是递归的, 因为第 3 步保证了每个计算都会终止。

由于我们这样设计语言 L , 因此对角化和自引用可以被用于为 M 能够被一个时间复杂性边界为 $f(n)$ 的图灵机接收的假设产生一个矛盾。设 M 是任意一个能够接收 L 的图灵机。那么在列举图灵机的某处可能出现 M , 也就是说 $M = M_i$ 。我们可以通过 L 中 u 的成员资格来获取自引用。因为 $L(M_i) = L$, M 接收 u_i 当且仅当 M 在 $f(n)$ 或更少的转换内停机, 或者 M 没能在前 $f(n)$ 个转换内停机。

通过矛盾证明 M 不以 f 为边界。假设 M 以 f 为边界并且设 $n = \text{length}(u)$ 。需要考虑两种情况: $u_j \in L$ 和 $u_j \notin L$ 。

如果 $u \in L$, 那么 M 在 $f(n)$ 或更少的转移内接收 u (因为假设 M_i 以 f 为边界)。但是, 正如前面指出的, M 接收 u_i 当且仅当 M 没有接收 u 或者 M 在 $f(n)$ 或更少的转换内停机。

如果 $u \notin L$, 那么 M_i 的计算在 $f(n)$ 个步骤内停机并且不接收 u_i 。在这种情况下, 根据 L 的定义有 $u_j \in L$ 。

在以上任意一种情况下, 假设 M 以 f 为边界都会导致矛盾。因此, 我们可以得出结论, 任意一个接收语言 L 的机器都不以 f 为边界。 ■

接下来我们介绍存在一种不存在能够接收它的最快的机器的语言。为了说明这种情况是如何发生的, 我们考虑一个机器的序列 N_1, N_2, \dots 都能接收同样来自 0^* 的语言。证明使用了函数 t , 其定义如下:

- i) $t(1) = 2$
- ii) $t(n) = 2^{t(n-1)}$

由此可得, $t(2) = 2^2, t(3) = 2^4, t(n)$ 是一系列 n 个 2 的指数。当运行输入 0^n 时, 表 14-6 在位置 $[i, j]$ 给出了机器 N_i 的转换次数。位置 $[i, j]$ 处的 $*$ 表示这个计算的转换次数是不相关的。 [453]

表 14.6 机器 N_i 及其计算

	λ	0	00	0^3	0^4	0^5	0^6	...
N_0	*	2	4	$t(3)$	$t(4)$	$t(5)$	$t(6)$	
N_1	*	*	2	4	$t(3)$	$t(4)$	$t(5)$	
N_2	*	*	*	2	4	$t(3)$	$t(4)$	
N_3	*	*	*	*	2	4	$t(3)$	
N_4	*	*	*	*	*	2	4	
\vdots								

如果机器存在这样一个序列, 那么

$$tc_{N_i}(n) = \log_2(tc_{N_{i+1}}(n))$$

对于所有 $n \geq i+1$ 都成立。因此, 我们有一个机器序列能够接收同样的语言, 并且序列中的每一个机器与其前趋相比都有一个更小的增长速度。定理 14.6.2 构造了一个语言, 这个语言具有“能够总是被更高效地接受”的属性。

在开始部分的讨论和定理 14.2.6 的构造中都是用到了, 当输入任意增大时, 增长速度能够度量函数性能的属性。根据表 14-6 中的模式可以看出, 我们仅仅在输入字符串长度为 $i+2$ 或者更大时才比较机器 N_i 和 N_{i+1} 的计算。

定理 14.6.2 存在一种语言 L , 使得对于任意接收 L 的机器 M , 都存在另外一个能够接收 L 并且时间复杂性为 $tc_{M'}(n) \in O(\log_2(tc_M(n)))$ 的机器 M' 。

设 t 是递归定义的函数, 如前所述, 当 $n > 1$ 时, $t(1) = 2$ 且 $t(n) = 2^{n-1}$ 。构造一个满足下面两个条件的递归语言 $L \subseteq \{0\}^*$ 。

1. 如果 M_i 接收 L , 那么对于所有大于 n_i 的 n , $tc_{M_i}(n) \geq t(n-i)$ 均成立。
2. 对于每个 k , 存在一个图灵机 M_j 使得 $L(M_j) = L$ 并且对于所有大于 n_k 的 n , $tc_{M_j}(n) \leq t(n-k)$ 均成立。

假设构造了一个满足上述条件的 L , 那么对于任意接收 L 的机器 M_i , 都存在一个同样接收 L 的机器 M_j 并且其时间复杂性

$$tc_{M_j}(n) \in O(\log_2(tc_{M_i}(n))),$$

设 $k = i+1$, 根据条件 2, 存在一个接收 L 的机器 M_j 且其时间复杂性 $tc_{M_j}(n) \leq t(n-i-1)$, 其中 $n \geq n_k$ 。然而, 根据条件 1

$$tc_{M_i}(n) \geq t(n-i) \quad n > n_i.$$

结合以上两个不等式和 t 的定义可得

$$tc_{M_i}(n) \geq t(n-i) = 2^{t(n-i-1)} \geq 2^{tc_{M_j}(n)}, \text{ 其中 } n > \max\{n_i, n_k\}.$$

也就是说, 对于所有的 $n > \max\{n_i, n_k\}$, $tc_{M_j}(n) \leq \log_2(tc_{M_i}(n))$ 均成立。

现在我们定义语言 L 的构造。按照顺序, 我们首先定义字符串 0^n ($n = 1, 2, \dots$) 是否属于语言 L 。在这个构造过程中, 图灵机列举序列 M_0, M_1, M_2, \dots 中的图灵机被标记为取消。在判定是否 $0^n \in L$ 时, 我们要检查一个图灵机 $M_{g(n)}$, 此处 $g(n)$ 是范围 $0, \dots, n$ 中的最后一个值 j 。因此

- i) M_j 之前从未被取消过,
- ii) $tc_{M_j}(n) < t(n-j)$ 。

当 $g(n)$ 没有定义时这样的值 j 也可能是不存在的。字符串 $0^n \in L$, 当且仅当 $g(n)$ 被定义并且 $M_{g(n)}$ 不接收 0^n 。如果 $g(n)$ 被定义了, 那么 $M_{g(n)}$ 被标记为取消。 L 的定义保证了一个取消的机器不能接收 L 。如果 $M_{g(n)}$ 被取消了, 那么 $0^n \in L$ 当且仅当 $M_{g(n)}$ 不接收 0^n 。因此, $L(M_{g(n)}) \neq L$ 。

定理 14.6.2 的证明包含了以下三个引理。第一个引理说明 L 是递归的, 后面两个引理说明 L 满足上述的条件 1 和条件 2。

引理 14.6.3 语言 L 是递归的。

证明: L 的定义提供了一种判定 $0^n \in L$ 是否成立的方法。如果 $g(n)$ 存在, 对于 0^n 的判定过程开

始于确定满足条件1和条件2的机器序列 M_0, \dots, M_n 中的第一个机器的索引 $g(n)$ 。为了做到这一点,在分析输入 $\lambda, 0, \dots, 0^n$ 时有必要确定序列 M_0, \dots, M_{n-1} 中的机器被取消。这就要求我们使用适当的 t 值比较表14-7中的函数复杂性。输入字母表包括独立的字符0,因此可以通过模拟输入为 0^n 时 M_i 的计算来确定 $tc_{M_i}(m)$ 。

表 14-7 确定取消机器的计算

输入	m	比较 $tc_{M_i}(m) \leq t(m-i)$
λ	0	$tc_{M_0}(0) \leq t(0-0) = t(0)$
0	1	$tc_{M_0}(1) \leq t(1-0) = t(1)$ $tc_{M_1}(1) \leq t(1-1) = t(0)$
00	2	$tc_{M_0}(2) \leq t(2-0) = t(2)$ $tc_{M_1}(2) \leq t(2-1) = t(1)$ $tc_{M_2}(2) \leq t(2-2) = t(0)$
\vdots	\vdots	\vdots
0^{n-1}	$n-1$	$tc_{M_0}(n-1) \leq t(n-1-0) = t(n-1)$ $tc_{M_1}(n-1) \leq t(n-1-1) = t(n-2)$ $tc_{M_2}(n-1) \leq t(n-1-2) = t(n-3)$ \vdots $tc_{M_{n-1}}(n-1) \leq t(n-1-(n-1)) = t(0)$

在记录了被取消的机器后,我们用输入为 0^n 的计算来确定 $g(n)$ 。从 $j=0$ 开始,如果 M_j 之前没有被取消,那么就计算 $t(n-j)$ 并模拟输入为 0^n 时 M_j 的计算。如果 $tc_{M_j}(n) \leq t(n-j)$,那么 $g(n)=j$ 。否则,增加 j 并循环执行比较直到找到 $g(n)$ 或者所有的机器 M_0, \dots, M_n 都被测试过。

如果 $g(n)$ 存在,使用输入 0^n 运行机器 $M_{g(n)}$ 。计算的结果能够确定 0^n 是否是 L 的成员: $0^n \in L$ 当且仅当 $M_{g(n)}$ 不接收 0^n 。上述过程描述了一个判定任意字符串 0^n 是否属于 L 的过程。因此, L 是递归的。 455

引理 14.6.4 L 满足条件1。

证明: 假设 M_i 接收 L 。首先,值得注意的是存在某个整数 p_i ,使得如果一个机器 M_0, M_1, \dots, M_i 从未被取消,那么取消它的优先级比前者字符串 0^n 的优先级要高。因为序列 M_0, M_1, \dots, M_i 中被取消的机器的数量是有所限制的,因此我们可以不知道 p_i 的值,但是它必然会发生,这也是我们需要知道的全部内容。

对于任意的 0^n ,其中 n 大于 p_i 和 i 的最大值,不存在可以被取消的 $M_k (k < i)$ 。假设 $tc_{M_i}(n) < t(n-i)$,那么在检查 0^n 时 M_i 将被取消。然而,一个被取消的图灵机不能接收 L 。因此对于所有 $n > \max \{p_i, i\}$ 都有 $tc_{M_i}(n) \geq t(n-i)$ 成立。 ■

引理 14.6.5 L 满足条件2。

证明: 我们必须证明,对于任意整数 k ,存在一个接收 L 的机器 M 并且 $tc_M(n) \leq t(n-k)$ 对所有大于某个 n_k 的 n 都成立。我们先从引理14.6.3描述的接收 L 的图灵机 M 开始。为了判定 0^n 是否属于 L ,机器 M 需要确定 $g(n)$ 的值并模拟 $M_{g(n)}$ 在输入 0^n 上的计算。为了获取 $g(n)$, M 必须确定在分析字符串 $\lambda, 0, \dots, 0^n$ 时 M_i 中的哪些被取消了。不幸的是,表14-7中给出的这些情况的直接度量需要多于 $t(n-k)$ 次转换。 456

正如引理14.6.4所述,当考虑到初始输入序列 $\lambda, 0, 00, \dots, 0^n$ 时,任意一个之前从未被取消的图灵机 $M_i (i \leq k)$ 会被取消。这个 p_i 值可用于推导前面所说的计算的复杂性。对于每一个 $m \leq p_k, 0^m$ 是否被接收的信息存储在接收 L 的机器 M 的状态中。

机器 M 在输入 0^n 上的计算可以分为两种情况:

第一种情况: $n \leq p_k$ 。 0^n 是否属于 L 仅通过 M 状态记录的信息来决定。

第二种情况: $n > p_k$ 第一个步骤是确定 $g(n)$, 通过模拟输入为 0^m ($m = k+1, \dots, n$) 时图灵机 M_i ($i = k+1, \dots, n$) 的计算来确定 M_i 是否在 0^n 上或者 0^n 之前被取消 因为 M_0, M_1, \dots, M_k 中的任何机器不会被输入长度大于 p_k 的输入取消, 所以我们只需要检查在范围 M_{k+1}, \dots, M_n 内的机器就可以了

跳过模拟 M_0, M_1, \dots, M_k 的能力能够减少度量输入字符串 0^n ($n > p_k$) 所需要的转换次数 表 14-7 给出的模拟数字被减少到如下表所示。

输入	m	比较 $tc_{M_i}(m) \leq t(m-i)$
0^{k+1}	$k+1$	$tc_{M_{k+1}}(k+1) \leq t(k+1 - (k+1)) = t(0)$
0^{k+2}	$k+2$	$tc_{M_{k+1}}(k+2) \leq t(k+2 - (k+1)) = t(1)$ $tc_{M_{k+2}}(k+2) \leq t(k+2 - (k+2)) = t(0)$
\vdots	\vdots	\vdots
0^n	n	$tc_{M_{k+1}}(n) \leq t(n - (k+1))$ $tc_{M_{k+2}}(n) \leq t(n - (k+2))$ \vdots $tc_{M_n}(n) \leq t(n - n) = t(0)$

检查当输入为 0^m 时 M_i 是否被取消最多需要 $t(m-i)$ 次转换 前面序列中任意一个计算所需要最多转换的情况发生在 $i = k+1$ 且 $m = n$ 时, 此时转换的最大次数是 $t(n-k-1)$ 。

机器 M 必须实施给出的每一个比较 模拟输入为 0^m 时 M 的计算最多需要 $t(n-k-1)$ 次转换 模拟后擦除带和准备后续的模拟在另外 $2t(n-k-1)$ 次转换内可以完成 模拟和比较循环必须在每一个机器 M_i ($i = k+1, \dots, n$) 的输入 0^m ($m = k+1, \dots, n$) 上重复 这样, 模拟过程最多重复 $(n-k)(n-k+1)/2$ 次 因此, M 所需要的转换次数少于 $3(n-k)(n-k+1)t(n-k-1)/2$ 次 也就是说,

$$tc_M(n) \leq 3(n-k)(n-k+1)t(n-k-1)/2.$$

然而, $3(n-k)(n-k+1)t(n-k-1)/2$ 的增长速度小于 $t(n-k) = 2^{(n-k-1)}$ 的增长速度 因此, $tc_M(n) \leq t(n-k)$ 对所有大于某个 n_k 的 n 都成立。 ■

上述的证明说明了对于任意接收 L 的机器 M , 都存在一个机器 M' 能够比 M 更高效地接收 L 现在 M' 接收 L , 所以, 再次通过定理 14.6.2 可知存在一个能够更高效地接收 L 的机器 M'' 这个过程可以不确定地持续下去, 从而产生一系列机器, 其中的每一个机器与其前驱机器相比, 都能以更小的增长速度接收 L 。

定理 14.6.2 揭示了算法计算的不直观的属性; 存在没有最优解的判定问题 给定一个问题的任意一个算法解, 都存在另外一个比它的效率明显提高的算法解。

14.7 计算机计算的模拟

我们关于算法复杂性的研究是基于图灵机在实现一个算法的计算时所需要的转换次数 然而, 我们所作的绝大部分计算工作并不在图灵机上而是在计算机上 为了说明图灵机计算分析的实际应用, 我们将在本节比较在一个标准计算机和一个图灵机上运行同一个算法的时间复杂性, 此处计算机计算的时间复杂性是通过计算过程中机器执行的指令数来度量的。

我们不会推导有关指令数和转换次数之间精确关系的定理 因为不同的计算机有不同的体系结构、指令系统、存储容量和计算能力, 所以推导这样的精确关系是不可能的 然而, 我们要做的是, 定义一种通用的、包含了标准机器或继承语言的机器指令 实际上, 我们赋予这种指令的弹性和计算能力比任何实际应用的计算机体系结构中的指令都要强。

首先值得注意的是, 我们研究的兴趣是一个实际的计算机而不是理论机器, 如图灵机 因此, 我们的机器必须有有限的存储, 存储可以想多大就多大, 但是是有限的 机器存储被划分为固定长度的有地址的字 在实际应用中, 一个字通常包括 32 或者 64 位, 但是在我们的机器中, 允许字的长度是任意固定的长度 关于字长惟一的约束是它要足够长以便能够容纳机器指令 每一个字都有一个相关

的数字地址用于检索和存储数据。

一个机器指令由说明要执行的操作的操作码和操作数构成。指令可以移动数据、执行算术或布尔运算、调整程序流或者分配额外的存储。临时的计算或动态增加计算中可用的存储可以请求存储分配。我们假设存在一个最大的存储,也就是说,一个独立指令的运行可以分配 m_w 个字。当然, m_w 这个数字可以跟我们希望的一样大。

操作数指明在什么位置检索数据,在什么位置存储结果,或者其他操作需要使用的地址。一个指令通常有一个或者两个操作数,但是我们允许每一个指令有最多 t 个操作数。因为 t 是可以显示地在指令中分配的最大地址数,因此我们假设一个指令的结果最多可以改变存储中的 t 个字。最后的约束(如果可以叫做一个约束的话)是指令集必须是有限的。

总结这些情况,我们能够考虑到一个体系结构和指令集满足以下条件的计算机的时间复杂性:

组 成	条 件
存储:	有限的
字大小:	固定字长,每个字包含 m_w 个字
指令集:	有限的
指令:	操作码和在一个字内的最多 t 个操作数
操作:	最多改变 t 个字,最多分配存储中的 m_w 个字

我们应该很清楚,大多数(如果不是全部的话)标准的计算机体系结构和指令集都满足这些根本的约束。至于某种具体的计算机体系结构的内存访问、指令执行和程序流维护的细节则不是我们关心的问题。我们只关心执行了多少指令。

现在,我们设计一个图灵机来模拟包括一系列指令的计算机。我们使用图 14-1 中给出的 $4+t$ 带图灵机模型, t 是指令中操作数的数量。程序和输入存储在带 1。像计算机存储一样,我们把带 1 划分为字:带的位置 0 到位置 m_w-1 组成字 0, m_w 到 $2m_w-1$ 组成字 1,依此类推。我们使用简单的内存分配机制:顺序地分配内存,并且内存一旦分配永远不会被释放。内存计数器记录带 1 存储种下一个空闲字的地址。

程序计数器包含下一个将要执行的指令的位置。除非一个指令声明使用下一个指令的位置作为他的一个操作数的值,程序控制是顺序的。输入计数器包含两个地址,即输入开始的位置和要读的下一个字的位置。另外一个计数带用于在带 1 上定位地址。最后,共有 t 各工作带,一个与指令的每个操作数相关的带。这些带可以被看作是图灵机等价的寄存器;只有相关的数据被转移到这些带上是数据上的操作才会执行。图 14-1 显示了我们的图灵机的配置。

现在,我们要推导图灵机在模拟一个计算的第 k 个指令的执行时所需要的转换次数的上界。一个指令可以取数据、存数据、分配存储和执行计算。我们的图灵机对指令执行的模拟包括以下几个动作:

- 针对操作要求的每一个操作数,装载其相关带上操作数 i 描述的数据;
- 执行给定的指令;
- 针对操作要求的每一个操作数,在操作数 i 指明的位置存储结果。

在第一步中,需要处理的数据可能在指令内,也可能指令仅仅包含了需要的数据的地址。

为了得到模拟一个指令的执行所需要的转换次数的上界,我们不切实际地假设每一个指令都执行了最大数量的每一种活动。也就是说,我们会计算每一个指令取 t 个字,执行一个操作,存储 t 个字,并且分配大小为 m_w 个字的存储多需要的转换次数。这样,我们需要确定每一个这样

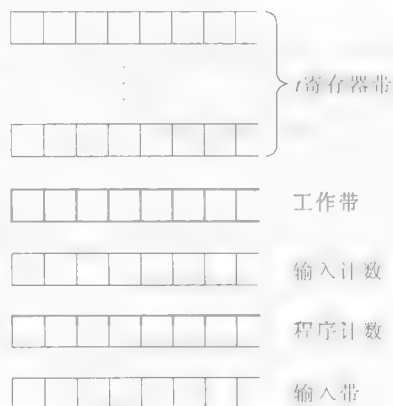


图 14-1 模拟计算机

的活动需要多少次转换。

[460]

由于指令的数量是有限的, 每一个指令最多使用 t 个操作数并且这些数据在寄存器带上的位置是已知的, 所以我们可以计算出执行任何操作所需要的最大转换次数。我们把这个数字记做 t_0 , t_0 仅仅取决于指令集并且与输入, 数据和一个计算的指令数量无关。

装载操作数和存储结果需要的转换次数依赖于图灵机正在使用的存储的大小。我们设存储指令需要 m_p 个位, 存储输入需要 m_i 个位, 并且在图灵机开始模拟指令 k 时总共分配的存储的大小是 m_k 。因此

$$m_k = m_p + m_i + k \cdot m_a$$

是图灵机在模拟第 k 个转换前分配的存储的最大数。

在模拟第 k 个转移的过程中, 图灵机可以在 m_k 个转换内定为任意的地址。为了找到一个字的开头, 图灵机把地址装载到计数带上。当地址不是 0 时, 程序带头向右移动 m_a 个方格并且减少计数带的值。当计数带的值为 0 时该过程终止, 此时程序带的带头正在读我们需要的数据的第一位。拷贝地址最多需要 m_a 次转换。因为该过程不会读最后一个字的位, 所以找到地址最多需要 $m_i - m_a$ 次转换。

所以, 模拟第 k 个指令的执行所需要的转换次数的上界是:

活动	找到指令	装载操作数	返回寄存器带头	执行操作	存储信息	返回寄存器带头
转换	m_k	$t \cdot m_k$	$t \cdot m_k$	t_0	$t \cdot m_{k+1}$	$t \cdot m_{k+1}$

由于操作可能分配额外的存储, 因此存储操作可能要访问 m_{k+1} 个带上的方格。累加与模型指令每一步相关的转换可以推导出模拟第 k 个指令的转移上界:

$$\begin{aligned} & (2t+1)m_k + 2tm_{k+1} + t_0 \\ &= (2t+1)(m_p + m_i + k \cdot m_a) + 2t(m_p + m_i + k \cdot m_a + m_a) + t_0 \\ &= (4t+1)m_p + (4t+1)m_i + 2t \cdot m_a + t_0 + (4t+1)k \cdot m_a \end{aligned}$$

[461]

值 m_p 、 m_a 和 t_0 是独立于输入的常量。如果当计算机的一个计算的输入长度为 $m_i = n$ 需要 $f(n)$ 个步骤, 那么我们的图灵机的模拟需要:

$$\begin{aligned} & \sum_{k=1}^{f(n)} ((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_0 + (4t+1)k \cdot m_a) \\ &= f(n)((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_0) + \sum_{k=1}^{f(n)} (4t+1)k \cdot m_a \\ &= f(n)((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_0) + (4t+1)m_a \sum_{k=1}^{f(n)} k. \end{aligned}$$

因此这个增长的速度大于 $O(nf(n))$ 或者 $O(f(n)^2)$ 。从计算机到图灵机的转换模拟多项式地增加了时间复杂性的阶。在实际应用中, 任何一个在计算机上能够在多项式时间内运行的算法都可以在多项式时间内被模拟成图灵机。

14.8 练习

1. 从表 14-3 中选择能够的描述一下函数的“最好”的大 O 。

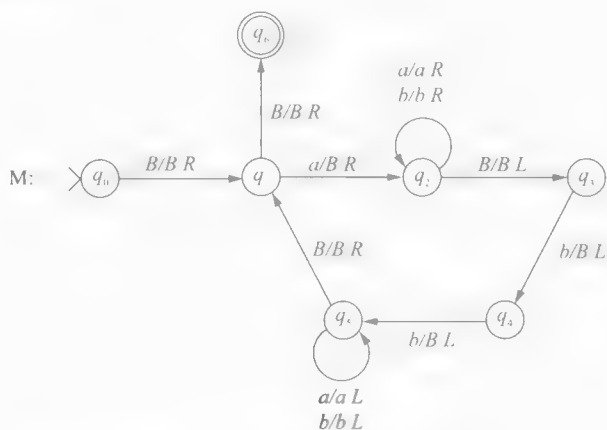
- $6n^2 + 500$
- $2n^2 + n^2 \log_2(n)$
- $\lfloor (n^3 + 2n)(n+5)/n^2 \rfloor$
- $n^2 \cdot 2^n + n!$
- $25 \cdot n \cdot \text{sqrt}(n) + 5n^2 + 23$

2. 设 f 是一个 r 次多项式。请证明 f 和 n^r 有同样的增长速度。

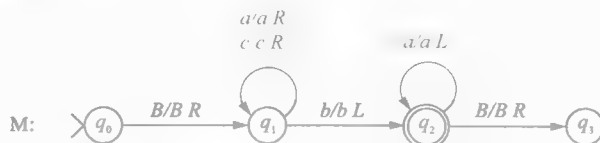
3. 使用 14.2.1 的定义或者限制规则证明以下的关系。

- $n \cdot \text{sqrt}(n) \in O(n^2)$

- b) $\log_2(n) \log_2(n) \in O(n)$
 c) $n' \in O(2^n)$
 d) $2^n \notin O(n')$
 e) $2^n \in O(n!)$
 f) $n! \notin O(2^n)$
4. $3^n \in O(2^n)$ 是否正确? 请证明你的答案。
5. 设 a 是一个大于1的自然数, c 是一个大于0的常量 $\log_a(n+c) \in O(\log_a(n))$ 是否正确? 请证明你的答案。
6. 设 $f(n) = n^{\log_2(n)}$
 a) 证明对于任何 $r > 0$ 有 $f(n) \notin O(n^r)$ 。也就是说, $f(n)$ 不是多项式边界的;
 b) 证明 $2^n \notin O(f(n))$ 。也就是说 $f(n)$ 不是指数级增长的。
7. 设 f 和 g 是两个一元函数, $f(n) \in \Theta(n')$ 并且 $g(n) \in \Theta(n')$ 给定与下列函数有相同增长速度的多项式大 Θ 并证明你的答案。
 a) $f+g$
 b) fg
 c) f^2
 d) $f \circ g$
8. 确定以下图灵机的时间复杂性。
 a) 例 8.2.1
 b) 例 8.6.3
 c) 例 9.1.2
 d) 例 9.2.1
9. 设 M 是下图所示的图灵机



- a) 输入 λ 、 a 和 abb , 跟踪 M 的计算。
 b) 描述输入长度为 n 的字符串时 M 的计算所需要的最大转换次数。
 c) 给出函数 tc_M 。
10. 设 M 是下图所示的图灵机



- a) 输入 abc 、 aab 和 cab , 跟踪 M 的计算。

- b) 描述输入长度为 n 的字符串时 M 的计算所需要的最大转换次数。
- c) 给出 $L(M)$ 的正则表达式。
- d) 给出函数 tc_M 。
11. 设 L 是 $\{a, b\}$ 上的语言, 如果 u 满足下述条件则 u 属于 L
- $u = a^i b^j$ 且 $length(u) \leq 100$, 或者
 - $length(u) > 100$
- 设计一个能够接收 L 的标准图灵机 M 。
 - 给出函数 tc_M 。
 - 能够描述时间复杂性函数 tc_M 的最好的多项式增长速度是多少?
12. 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个接收语言 L 的双带图灵机。设 N 是根据定理 14.4.2 构造的机器 ($m = 12$), 请求出 N 的带字母表的大小和状态数。
13. 设计一个能够接收语言 $\{a^i b^j \mid i \geq 0\}$ 且时间复杂性 $tc_M \in O(n \log_2(n))$ 的标准图灵机。提示: 每次经过一个数据时, 标记一半以前没有标记过的 a 和一半以前没有标记过的 b 。

参考文献注释

我们已经介绍了一个计算在时间方面的复杂性。第 17 章将研究时间和空间复杂性的关系。Blum [1967] 介绍了一种度量抽象复杂性的公理化方法, Hartmanis 和 Hopcroft [1971] 进一步发展和完善了该方法。在 1985 年的美国计算机协会 (ACM) 图灵奖演讲文章中, Richard Karp [1986] 描绘了关于复杂性理论的最初发展和方向的有趣的个人历史。

第 15 章 \mathcal{P} 、 \mathcal{NP} 和库克定理

复杂性理论是用于确定判定问题在理论上是不是可解的。在复杂性理论中,我们进一步把可解问题划分为有实际解的问题和只是在理论上可解的问题。问题在于,理论上可解的问题可能没有实际解;可能没有算法可以在不需要额外时间或存储的情况下就能解决问题。不存在有效算法的问题称为难解的。由于时间复杂性增长的速度,非多项式的算法不能被看作是对问题的所有情况都可行的,而是仅对于问题的最简单情况是可行的。将可解的判定问题类划分为多项式时间问题和非多项式时间问题,其主要目的是把高效的可解问题与难解问题区分开来。

有许多著名的问题都有多项式时间的不确定型解,但是它们没有多项式时间的确定型解。本章我们将揭示使用确定型多项式时间算法的可解性与使用非确定型多项式时间算法的可解性之间的关系。是否每一个使用不确定型算法在多项式时间内可解的问题在使用确定型算法时在多项式时间内也是可解的,目前仍然是理论计算机科学不能回答的一个著名的开放性问题。

可解的判定问题和递归语言之间的二元性使得我们能够以递归语言的方式定义复杂性类。因为时间复杂性将输入字符串的长度与转换的次数关联起来,因此判定问题实例表示的选择可能会改变算法的复杂性。为了将这种表示的影响与问题固有的难度区分开,我们将在表示中引入一些简单的约束,这样,表示的改变只能在多项式时间内影响解的复杂性。

(465)

15.1 非确定型图灵机的时间复杂性

非确定型计算与其对应的确定型计算有本质的不同。确定型机器在搜索解的过程中会生成和检查多种可能性;而非确定型机器则使用猜想和检查策略,因而只需要确定是否一个可能性能够提供解。考虑判定一个自然数 k 是否是合数(非素数)的问题。一个构造性确定型解可以通过顺序地检查每一个从 2 到 \sqrt{k} 之间的数字是否是 k 的因子来获取。如果发现一个因子,那么 k 就是合数。一个不确定型的计算开始于从指派的范围任意选择一个值。如果猜想是一个因子那么一个除法运算就可以确定。如果 k 是合数,那么其中的一个不确定选择会产生一个因子并且计算会返回肯定的回答。

如果至少有一个计算能够终止在接收状态,那么这个字符串就被一个非确定型机器接收。字符串的接收性不会被其他在不接收状态停机或不停机的计算的存在所影响。然而,一个算法最坏情况下的性能度量了整个计算的效率。

定义 15.1.1 设 M 是一个非确定型图灵机。 M 的时间复杂性函数是 $tc_M: \mathbb{N} \rightarrow \mathbb{N}$, 它使得当输入字符串长度为 n 时,通过任意转换的选择,一个计算处理的转换的最大次数是 $tc_M(n)$ 。

上述定义与确定型机器的时间复杂度的定义是相同的。该定义强调了不确定性分析必须考虑对一个输入字符串的所有可能的计算。与确定型机器的情况相同,我们的时间复杂度定义假设 M 的每一个计算都能终止。

使用猜测和检查策略的非确定型计算通常比与其对应的确定型计算简单。这种简化减少了一次单独计算所需要的转换的次数。使用这个策略,我们可以构造一个能够接收 $\{a, b\}$ 上的回文的非确定型机器。

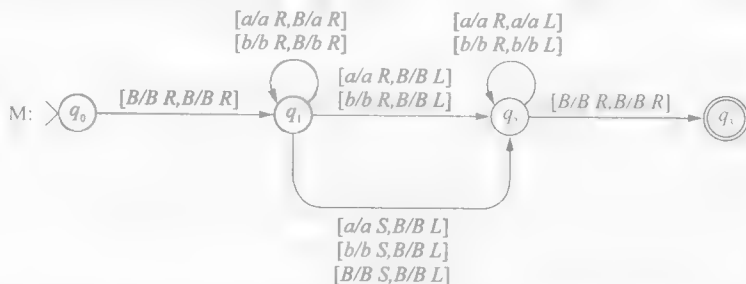
例 15.1.1 双带非确定型图灵机 M 能够接收 $\{a, b\}$ 上的回文。当输入被拷贝到带 2 上时,两个带的带头都向右移动。始于 q_1 的转换“猜想”字符串的中心。将带 1 的带头右移并将带 2 的带头左移的始于 q_1 的转换是对奇数长度回文的检查,在同样的位置离开带 1 的带头的转换是对偶数长度回文的检查。如果带 1 和带 2 同时读到空白接收计算就停机,此时转换次数最大。时间复杂性

$$tc_M(n) = \begin{cases} n+2 & \text{如果 } n \text{ 是奇数} \\ n+3 & \text{如果 } n \text{ 是偶数} \end{cases}$$

说明偶数长度字符串的接收需要额外的转换。

□

466



第8.7节介绍的从非确定型机器到确定型机器的转换所用到的策略不能够保持多项式时间可解性。然而，它却提供了能够接收原非确定型机器语言的确定型机器的时间复杂性的上界。

定理 15.1.2 设 L 是一个能被单带非确定型图灵机 M 以时间复杂性 $tc_M(n) = f(n)$ 接收的语言。那么 L 可以被一个确定型图灵机 M' 以时间复杂性 $tc_{M'}(n) \in O(f(n)c^{f(n)})$ 接收，其中 c 是状态和 M 中符号对转换的最大次数。

证明： 设 $M = (Q, \Sigma, \Gamma, \delta, q_0)$ 是一个能够对所有输入停机的单带非确定型图灵机，设 c 是状态和 M 中符号对转换的最大次数。我们通过将 M 的一个唯一的计算与一个序列 (m_1, \dots, m_n) （其中 $1 \leq m_i \leq c$ ）关联起来从而实现从非确定性到确定性的转换。 m_i 的值表示 M 的 c 个可能的转换中，哪个转换应该在计算的第 i 步执行。

在第8.7节我们介绍了一个三带确定型图灵机 M' ，当输入为 w 时，它的计算能够迭代地模拟 M 输入为 w 时的所有可能计算。对于每一个长度为 n 的输入， M 的任意一个计算的最大转换次数为 $f(n)$ 。为了模拟 M 的一个单独的计算，机器 M'

1. 生成一个整数序列 (m_1, \dots, m_n) ，其中 $1 \leq m_i \leq c$ ；
2. 模拟序列 (m_1, \dots, m_n) 描述的 M 的计算；并且
3. 如果计算不接收这个字符串，那么 M' 回到步骤 1 继续执行。

在最坏的情况下需要检查 $c^{f(n)}$ 个序列。 M' 可以使用 $O(f(n))$ 个转换来模拟 M 的一个独立计算。因此， M' 的时间复杂性是 $O(f(n)c^{f(n)})$ 。

467

定理 15.1.2 介绍的时间复杂性 $O(f(n)c^{f(n)})$ 是从 M' 到 M 的特殊构造的制品。其他考虑到问题中特定语言的属性的方法也可以用于设计确定型图灵机，并且其时间复杂性比定理 15.1.2 给出的上界小很多。例如，当处理一个长度为 n 的输入字符串时，例题 8.7 中能够接收 $(a \cup b \cup c)^*(abc \cup cab)(a \cup b \cup c)^*$ 的非确定型机器最多使用 $n+3$ 个转换。定理 15.1.2 使用的构造方法生成一个能够以时间复杂度 $\Theta(n \cdot 3^n)$ 接收语言的确定型机器。然而，这个语言同样也可以被标准图灵机以时间复杂度 $n+1$ 接受。

后面几节的内容将分析介绍能够在多项式时间内被确定地解出的问题和能够在多项式时间内被非确定地解出的问题之间的关系。

15.2 P类和NP类

设 L 是 Σ^+ 上的一个语言，如果存在一个能够确定 L 的成员资格的算法，计算所要求的时间至多会随着输入串长度增长而多项式地增长，那么说 L 在多项式时间内是可判定的或者简称为多项式的。多项式时间可判定性的概念是通过标准图灵机的转换定义的，它能够用于度量计算的时间。

定义 15.2.1 一个语言 L 是多项式时间可判定的，如果存在一个能够以 $tc_M \in O(n^r)$ 接收 L 的标准图灵机 M ，其中 r 是一个与 n 无关的自然数。多项式时间可判定的语言的家族表示为 P 。

\mathcal{P} 类以在标准图灵机上的算法应用的时间复杂性的形式来定义。我们可以像选择算法度量的计算模型那样，简单地选择一个多道、多带或者双向确定型图灵机。在为分析而选定的确定型图灵机模型的选择下，多项式可判定语言或可解的判定问题的 \mathcal{P} 类是不变的。第14.4节介绍了可以被一个多道机器在时间 $O(n')$ 接收的语言也可以被一个标准图灵机在时间 $O(n')$ 接收。从多带图灵机到标准图灵机的转换保持了多项式解。一个可以被多带机器在 $O(n')$ 接收的语言可以被标准机器在时间 $O(n^2)$ 内接收。

第14.7节分析了在计算机上运行程序的复杂性和在图灵机上对其进行模拟的复杂性之间的关系。与计算机执行的指令数相比，图灵机模拟的转换次数仅有多项式的增加。因此任何一个我们认为在标准计算机上多项式可解的问题均是在 \mathcal{P} 中的。处于机器和体系结构变化中的 \mathcal{P} 类的鲁棒性为选择其定义可解问题和难解问题的边界提供了支持。

解决判定性问题的非确定型机器的计算检查问题的一个可能解。非确定地选择一个潜在的解而不是系统的解决所有可能的解的能力降低了非确定型机器的计算的复杂度。与定义 \mathcal{P} 类的方法相同，我们可以定义在多项式时间内被非确定型图灵机接收的语言的家族。

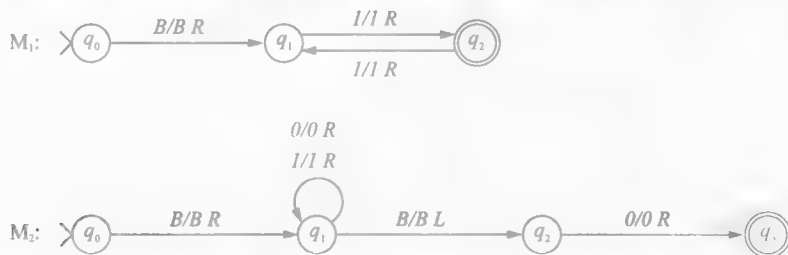
定义 15.2.2 一个语言 L 被称为能够在非确定的多项式时间 (nondeterministic polynomial time) 内被接收，如果存在一个能以 $tc_M \in O(n^r)$ 接收 L 的非确定型图灵机 M ，其中 r 是一个与 n 无关的自然数。可以在非确定的多项式时间内被接收的语言的家族表示为 \mathcal{NP} 。

\mathcal{NP} 家族是递归语言的一个子集；转换次数的多项式边界保证类 \mathcal{M} 的所有计算最终都能终止。因为每一个确定型机器也是一个不确定性机器，所以 $\mathcal{P} \subseteq \mathcal{NP}$ 。相反结论的状态是本章后续内容的话题。

15.3 问题表示和复杂性

用于解决判定问题的图灵机的设计包括两个步骤：首先是问题实例的字符串表示，其次是设计分析结果字符串和解决问题的机器。在判定性的学习中，我们唯一关心的是有没有考虑到解决问题的算法的发现以及计算需要的资源。因为图灵机的时间复杂性把输入的长度和计算中转换的次数关联起来了，所以表示法的选择可以对计算需要的工作量有重要影响。

在第11章中，我们设计了两个简单的用于解决一个自然数是否是偶数的判定问题的图灵机。机器 M_1 的输入使用自然数的一元表示法，机器 M_2 的输入使用二进制表示法：



这两个机器的时间复杂性是线性的，并且表示法的不同没有显著地影响复杂性。不幸的是，情况并不总是这样。机器 M_1 的改动会给复杂性带来明显的影响。

我们可以构造一个图灵机 T 来将用二进制表示的自然数转换为一元表示的自然数（练习6）。 T 和 M_1 的顺序操作生成 M_3 。



M_3 是偶数问题的另外一个解决方案。让我们来看一下这个方案的复杂性。下面的表给出了二进制表示到一元表示的转换导致的字符串长度的增加。第二列给出了第一列中字符串长度的最大二进制

[468]

[469]

数,最后一列是相应的一元表示。

字符串长度	最大二进制数	十进制值	一元表示
1	1	1	$11 = 1^2$
2	11	3	$1111 = 1^4$
3	111	7	$11111111 = 1^8$
i	1^i	$2^i - 1$	1^{2^i}

M_1 的时间复杂性是由 T 和 M 的复杂性决定的。对于每个长度为 i 的输入,字符串 1^i 要求 M_1 最大次数的转换。 M_1 的时间复杂性是:

$$\begin{aligned} tc_{M_1}(n) &= tc_T(n) + tc_{M_1}(2^n) \\ &= tc_T(n) + 2(2^n) + 2, \end{aligned}$$

及时转换没有添加额外的工作,这个复杂性也是指数级的。 M_1 回答这个问题的策略没有改变;之所以会有时间复杂性的增加是因为使用二进制表示减小了输入字符串的长度。

下面假设的情况进一步说明了在评估判定问题的时间复杂性时表示法的重要性。假设一个问题 P , 它的实例使用字母表 Σ 上的字符串表示,该问题可以被图灵机 M 以时间复杂性 $tc_M(n) = 2^n$ 解决。我们可以按照下述方法为 P 构造另外一个表示法: 在字母表中添加一个新的字符 $\#$, 如果一个问题实例在原表示法中被表示为长度为 n 的字符串 w , 那么在新的表示法中它被表示为 $w\#^n$ 。我们可以根据 M 得到一个解决问题 P 的机器 M' 。除了 M' 处理 $\#$ 的方式与 M 处理空白的方式一样以外, M' 的计算与 M 的计算也相同。由于输入字符串长度的增加,所以 $tc_{M'}(n) = n$ 。

470

上述例子提供了一种操作时间复杂性函数的方法,以便人工地将低效的算法改为高效的算法。如果可以改变输入字符串的长度而不改变下面的计算,那么在时间复杂性函数上就存在一个相应的缩减。

时间复杂性对表示的大小的依赖说明对复杂性分析而言,不是每一个表示法都是可接收的。使用最小的表示可以避免表示长度对复杂性的影响。然而,这样一个需求过于严格也没有必要。我们介绍多项式时间转换的概念是为了非形式化地描述表示法对复杂性分析的适宜性条件。

有实例 p_0, p_1, p_2, \dots 的判定问题 P 的表示法是一个从问题实例到 Σ 上的字符串的映射 rep , 其中 $rep(p_i)$ 是 p_i 的表示法。设 rep_1 和 rep_2 分别是 P 在字母表 Σ_1 和字母表 Σ_2 上的表示法。表示法 rep_1 是多项式时间可转换为 rep_2 的,如果存在一个函数 $t: \Sigma_1^* \rightarrow \Sigma_2^*$ 使得:

- i) 对于所有的 i 都有 $t(rep_1(p_i)) = rep_2(p_i)$;
- ii) 如果 $u \in \Sigma_1^*$ 不是问题实例的表示法,那么 $t(u)$ 也不是 Σ_2^* 中问题实例的表示法;且
- iii) t 是标准图灵机 T 在多项式时间内可计算的。

如果 rep_1 在多项式时间可转换为 rep_2 , 那么相对于 $rep_1(p_i)$ 的长度, $t(rep_1(p_i))$ 的长度不可能以快于多项式的方式增长;可以添加到表示法中的字符的数目必然少于 T 的转换次数。

现在,假设 P 是图灵机 M 使用表示法 rep_2 在多项式时间内可解的。 T 和 P 的顺序组合



能够产生一个使用表示法 rep_1 的多项式时间解。因此,仅仅有多项式差别的表示法之间的不同不会影响到问题的可解性。一个问题的最合理的表示法仅在长度上与最小表示法有多项式的差别。这种情况的一个明显的例外是使用一元表示法的自然数,与二元表示法的输入字符串长度相比,此时输入字符串长度成指数级增加。正是这个原因,自然数复杂性分析永远是用二进制表示法。根据这一点,我们使用符号 \bar{i} 表示数字 i 的二进制形式。

根据以上描述的方针,一个使用一元表示法表示自然数有多项式解但是使用二进制表示法表示自然数没有多项式解的问题不被认为是多项式时间可解的。这个属性的问题被叫做伪多项式的 (pseudo-

polynomial), 因为对那些没有意识到表示法影响的人而言, 在分析判定问题的复杂性时, 使用一元表示法的解看上去是多项式时间解。 [471]

15.4 判定问题和复杂性类

本节我们列出了几个 \mathcal{P} 类和 \mathcal{NP} 类中的判定问题。我们不会描述解决这些问题的算法细节, 因为前面的内容已经给出了它们的解或者后续几章将会详细介绍他们的解法。这个列表的目的是从每个类中提供一些熟悉的问题的例子, 以便识别出同一个类中解决问题的算法共有的属性。

回文的接收性问题

输入: 字母表 Σ 上的字符串 u

输出: 是; 如果 u 是一个回文

否; 其他

复杂性: 在 \mathcal{P} 内——是

有向图的路径问题

输入: 图 $G = (N, A)$, 节点 $v_i, v_j \in N$

输出: 是; 如果在 G 中从 v_i 到 v_j 存在一条路径

否; 其他

复杂性: 在 \mathcal{P} 内——是

乔姆斯基范式语法中的可导性

输入: 乔姆斯基范式语法, 字符串 w

输出: 是; 如果存在一个推导 $S \Rightarrow^* w$

否; 否则

复杂性: 在 \mathcal{P} 内——是

上述的每一个问题都有多项式时间解。正如第 14.3 节介绍的, 回文可以被标准图灵机以时间复杂性 $O(n^2)$ 接受。迪杰斯特拉 (Dijkstra) 的算法可在 $O(n^2)$ 时间内发现两个节点之间的路径 (如果存在这样的路径), 其中 n 是图中节点的个数。第 4.6 节介绍的 CYK 算法可用 $O(n^3)$ 个步骤完成动态编程表, 从而确定由乔姆斯基范式语法定义的语言的成员资格。

可满足性

输入: 合取范式形式的布尔公式 u

输出: 是; 如果一个为真的赋值满足 u

否; 否则

复杂性: 在 \mathcal{P} 内——未知

在 \mathcal{NP} 内——是 [472]

哈密尔顿回路问题

输入: 有向图 $G = (N, A)$

输出: 是; 如果存在一个能够正好访问 G 的所有边一次的简单回路

否; 否则

复杂性: 在 \mathcal{P} 内——未知

在 \mathcal{NP} 内——是

和的子集问题

输入: 集合 S , 函数 $v: S \rightarrow \mathbb{N}$, 数字 k

输出: 是; 如果存在一个 S 的子集 S' , 且 S' 的总和是 k

否; 否则

复杂性: 在 \mathcal{P} 内——未知在 \mathcal{NP} 内——是

我们使用猜想和检查策略可以较容易地以非确定方式解决上述的每一个问题。可满足性问题的猜想是一个独立的真值赋值。根据公式的长度, 可以在多项式时间内验证一个特定的真值赋值是否满足合取范式形式的公式。哈密尔顿 (Hamiltonian) 回路问题的猜想是构成一个包含 $n+1$ 个节点的序列, 验证过程检查这个序列是否定义了图的一次周游。相似的, 和的子集问题的猜想是一个子集, 检查仅仅是累加子集中的值。

对那些对 \mathcal{P} 而言是未知的问题来说, 确定型算法通常不能提供对问题本质的洞察力, 而只是执行一个穷举搜索。下一节我们会给出这个问题的说明, 同时我们介绍哈密尔顿回路问题的确定型解法和非确定型解法。

我们再给出一个前面介绍过的问题, 它不属于复杂性类。这个问题考虑的是确定由正则表达式描述的、可以包含用 u^* 当作 uu 的缩写的语言。例如, $(a^*)^*b(a \cup b)^*$ 表示所有这样的字符串, 这些字符串中, 在首个 b 出现之前有偶数个 a 。

带平方的正则表达式

输入: 字母表 Σ 上的正则表达式 a 输出: 是; 如果 $a \neq \Sigma^*$

否; 否则

复杂性: 在 \mathcal{P} 内——否在 \mathcal{NP} 内——否

[473] 在介绍了空间复杂性后, 我们还会介绍这个问题的任何一个解法所需要的空间和时间都与正则表达式的长度成指数级的增长。

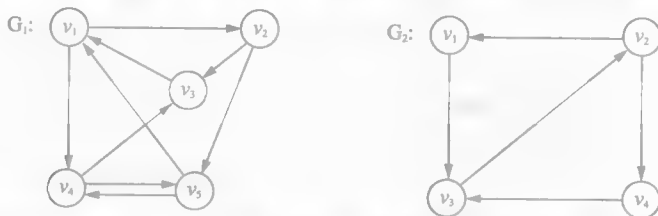
15.5 哈密尔顿回路问题

我们在这里介绍哈密尔顿回路问题是为了说明判定问题的确定型解和不确定型解在策略和复杂性两方面的差异。首先我们对前面介绍的问题给出一个更详细的描述。

设 G 是 n 个节点的有向图, 其节点编号从 1 到 n 。哈密尔顿回路是 G 中满足以下条件的路径 i_0, i_1, \dots, i_n :

i) $i_0 = i_n$ ii) 当 $i \neq j$ 时, $i_i \neq i_j$ ($0 \leq i, j < n$)

也就是说, 哈密尔顿回路是一条访问每个节点一次且仅一次、终止于起始点的路径。一个哈密尔顿回路通常叫做一个周游 (Tour)。例如, 图 G_1 的一个周游是 $v_1, v_2, v_3, v_4, v_5, v_1$, G_2 没有周游。



哈密尔顿回路问题是要判定一个有向图是否有一个周游。因为每一个节点都包含在周游内, 所以我们假设每一个周游都开始并结束于节点 1。

例 15.5.1 的确定型解完全彻底地搜索节点序列以判定每一个序列是否是周游。该方法系统地生成和测试了节点序列直到发现一个周游或者所有的可能都检查完毕为止。删除上述的生成和测试循环可以得到非确定型解。一个非确定型猜测生成一个节点序列, 并使用与确定型计算相同的过程检查其子序列。

例 15.5.1 在本例题中, 我们介绍一个解决哈密尔顿回路问题的四带确定型图灵机的活动。第一步是为节点编号从 1 到 n 的有向图设计表示法。表示法的字母表是 $\{0, 1, \#\}$, 并且使用节点编号的二进制形式表示节点。一个有 n 个节点 m 条边的图可以表示为以下的输入字符串

$$\bar{x}_1 \# \bar{y}_1 \# \# \cdots \# \bar{x}_m \# \bar{y}_m \# \# \# n,$$

其中 $[x_i, y_i]$ 是图的边, \bar{x} 是数字 x 的二进制形式。

[474]

在整个计算中, 带 1 用于维护边的表示。该计算能够生成和检查 $n+1$ 个节点的序列 $1, i_1, \dots, i_{n-1}, 1$ 以便确定它们是否组成一个周游。计算在带 2 上按照顺序生成序列。序列 $1, n, \dots, n, 1$ 的表示记录在带 4 上并且用于触发停机条件。图 8-1 中图灵机使用的技术可用于在带 2 上生成序列

一个计算是满足下列条件的循环:

1. 在带 2 上生成一个序列 $B \bar{1} B \bar{i}_1 B \bar{i}_2 B \cdots B \bar{i}_{n-1} B \bar{1} B$,
2. 如果带 2 和带 4 相同则停机, 并且
3. 检查序列 $1, i_1, \dots, i_{n-1}, 1$, 如果该序列是一个周游则停机。

如果计算在第二步停机, 那么所有检查过的序列以及这个图都不包含哈密尔顿回路。

第三步的分析开始于机器配置

带 4 $B \bar{1} (B \bar{n})^{n-1} B \bar{1} B$

带 3 $B \bar{1} B$

带 2 $B \bar{1} B \bar{i}_1 B \cdots B \bar{i}_{n-1} B \bar{1} B$

带 1 $B \bar{x}_1 \# \bar{y}_1 \# \# \cdots \# \bar{x}_m \# \bar{y}_m B \# \# \# n B$ 。

我们依次被检查节点 i_1, \dots, i_{n-1} 。满足下列条件时, 节点 i_j 将被添加到带 3 的序列中:

- i) $i_j \neq 1$;
- ii) $i_j \neq i_k$ ($1 \leq k \leq j-1$); 并且
- iii) 带 1 上有边 $[i_{j-1}, i_j]$ 。

也就是说, 如果 $1, i_1, \dots, i_j$ 是图中的一个非循环路径, 那么 i_j 被加入进来。如果带 2 上序列的每一个节点 i_j ($j = 1, \dots, n-1$) 都被添加到带 3 并且存在一条从 i_{n-1} 到 1 的边, 那么带 2 上的路径就是一个周游并且计算接收该输入。

当输入图不包含一个周游时, 该计算会检查和拒绝每一个 $1, i_1, \dots, i_{n-1}, 1$ 序列。对于一个有 n 个节点的图, 存在 n^{n-1} 个这样的序列。除了检查序列相关的计算, 序列的数量随着图的节点数做指数级增长。由于我们使用二进制形式为节点编码, 因此, 如果节点的数目增加到 $2n$ (但是图中边的数量不增加), 那么输入字符串的长度就增加一个字符。所以, 增加输入的长度会导致必须被检查的潜在序列数量的指数级增长。□

我们已经介绍过哈密尔顿回路问题在指数时间内是可解的。这个问题不属于多项式时间内也不能解决的范畴。到目前为止, 还没有发现多项式时间算法。这也许是因为不存在这样的解或者也许是因为我们不够聪明所以没有发现! 发现多项式时间解法的可能性和分支是本章后续内容的主要话题。

[475]

使用猜想和检查策略的不确定性计算通常比他们对应的确定性计算简单。这种简化减少了计算所需要的转换次数。人们通常构造使用这种策略的不确定型图灵机来在多项式时间内解决哈密尔顿回路问题。

例 15.5.2 我们可以通过修改例 15.5.1 中的确定型图灵机来构造一个在多项式时间内解决哈密尔顿回路问题的三带非确定型图灵机。非确定型图灵机不再需要第四条带, 在确定型图灵机中, 当图不包含周游时这条带用于结束计算。计算

1. 停机并且拒绝输入, 如果图中包含少于 $n+1$ 条边,
2. 在带 2 上非确定性地生成序列 $1, i_1, \dots, i_{n-1}, 1$, 并且
3. 利用带 1 和带 3 来确定带 2 上的序列是否定义了一个周游。

为了说明非确定型图灵机是多项式的, 我们为计算中的转换次数构造一个上界。当节点序列定义一个周游时转换次数达到最大。否则, 计算终止时在带 2 上检查的边数少于 $n+1$ 。由于节点是用二进

制形式表示的, 所以带需要对任意一个节点编码的最大次数是 $\lceil \log_2(n) \rceil + 1$ 。

最坏情况下的性能在当图包含多于 $n+1$ 条边时发生。包含更少边的图的计算在第 1 步停机, 从而避免了第 3 步检查所需要的转换。因此, 算法最坏情况下性能的输入长度依赖于图中边的数量。设 k 是边的条数。我们将说明转换次数的增长速度是 k 的多项式。由于输入长度不能比 k 增长得更慢 (每条边在带上至少需要三个位置), 所以其时间复杂性是多项式的。

在第 1 步拒绝输入要求计算比较输入中边的数量和节点的数量。这项工作可以在随边数多项式地增长的时间内完成。

如果计算不在第 1 步停机, 那么我们可以得知边的数量大于节点的数量。在带 2 上生成猜测并重新布置带头的位置需要 $O(n \log_2(n))$ 次转换。现在我们假设带 3 包含带 2 上的初始序列 $B \# i_1 \# \dots \# i_n$, 后续的计算包含一个循环:

[476]

1. 移动带 2 和带 3 的带头至带 3 的第一个空白的位置 ($O(n \log_2(n))$ 次转换),
2. 检查带 2 上已经编码的节点是否已经在带 3 上 ($O(n \log_2(n))$ 次转换),
3. 检查从 i_1 到 i_n 是否存在一条边 ($O(n \log_2(n))$ 次转换检查所有边并重新布置带头的位置), 并且
4. 在带 3 上写 i_i 并重新布置带头的位置 ($O(n \log_2(n))$ 次转换)。

一个计算包括在带 2 上生成序列, 随后是检查序列。检查序列的循环一直重复执行直到带 2 上的每一个节点 i_1, \dots, i_n 都被检查。步骤 3 的重复会导致整个计算中转换的次数以速度 ($O(k^2 \log_2(k))$) 增长。

非确定型机器时间复杂度的增长速度是由计算时在边列表中搜索某个特定边的过程决定的。这与确定型机器不同, 确定型机器对 n 个节点的整个序列集合的穷尽搜索决定了其增长速度。□

15.6 多项式时间归约

语言 L 到语言 Q 的归约把 L 的成员资格问题转换为 Q 的成员资格问题。归约在语言可判定性证明中扮演重要角色, 并且与分类问题中的可解性有同样重要的作用。设 r 是机器 R 计算的从 L 到 Q 的归约。如果 Q 被机器 M 接收, 那么 L 也会被下面这样的机器接收:

- i) 在输入字符串 $w \in \Sigma_1^*$ 上运行 R , 并且
- ii) 在 $r(w)$ 上运行 M 。

字符串 $r(w)$ 被 M 接收, 当且仅当 $w \in L$ 。在复杂性分析中, L 成员资格问题复合解的时间复杂性分析包括转换 L 的实例所需要的时间和获得 Q 的解所需要的时间。因为我们把高效可解的问题和多项式时间复杂性看作是等价的, 因此用同样的条件要求一个归约的时间复杂性似乎是合理的。

定义 15.6.1 设 L 和 Q 分别是字母表 Σ_1 和字母表 Σ_2 上的语言。我们称 L 对 Q 是多项式时间可归约 (reducible in polynomial time), 如果存在一个多项式时间的计算函数 $r: \Sigma_1^* \rightarrow \Sigma_2^*$ 使得 $w \in L$ 当且仅当 $r(w) \in Q$ 。

多项式时间归约非常重要, 因为归约的转换的边界限制了后续机器输入字符串的长度。这个属性保证了多项式时间归约和多项式时间算法的结合能够产生另外一个多项式算法。

[477]

定理 15.6.2 设 L 是在多项式时间内归约到 Q 的语言且 $Q \in \mathcal{P}$, 那么 $L \in \mathcal{P}$ 。

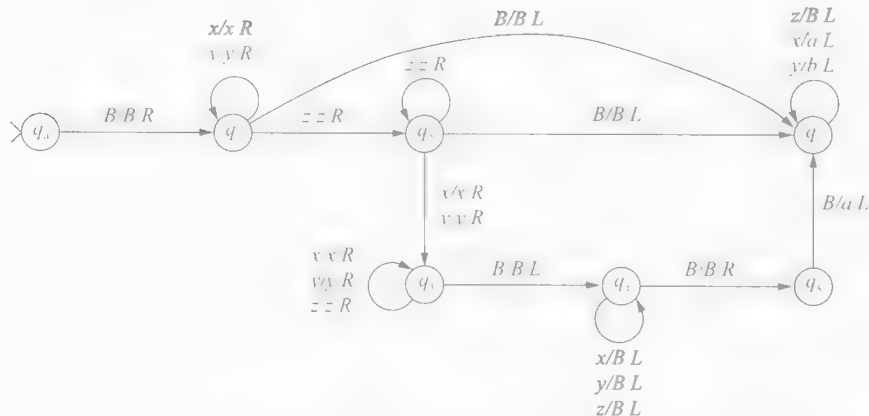
证明: 如前所述, 我们设 R 表示计算从 L 到 Q 的归约的机器, M 是决定 Q 的机器。 L 能被顺序运行 R 和 M 的机器接收。时间复杂性 tc_R 和 tc_M 结合以生成复合机器的计算的转换次数上界。输入为 w 时, R 的计算生成字符串 $r(w) \in \Sigma_2^*$, 它同时也是 M 的输入。函数 tc_R 可以用于说明输入长度为 $r(w)$ 时的边界。如果 R 的输入字符串 w 的长度为 n , 那么 $r(w)$ 的长度不可能超过 n 和 $tc_R(n)$ 的最大值。

M 的计算最多处理 $tc_M(k)$ 次转换, 其中 k 是其输入字符串的长度。复合机器的转换次数以两个独立计算估计的和为边界。如果 $tc_R \in O(n^r)$ 且 $tc_M \in O(n')$, 那么

$$tc_R(n) + tc_M(tc_R(n)) \in O(n^{r'}).$$

■

例 15.6.1 图灵机 R



将语言 $L = \{x^k y^l z^k \mid k \geq 0, l \geq 0\}$ 归约到 $Q = \{a^i b^i \mid i \geq 0\}$ 。11.3 节已经介绍了这个归约的动机。现在我们关心的是它的时间复杂性分析。

对于长度为 0 和 1 的字符串, $tc_R(0) = 2$ 且 $tc_R(1) = 4$ 。当一个 z 后有一个 x 或者 y 时, 对字符串后续部分的最坏情况下的计算得以发生。在这种情况下, 输入字符串在状态 q_1 、 q_2 和 q_3 读入, 在状态 q_3 被擦除。机器可以通过在输入位置写 a 来完成计算。时间复杂性是 $tc_R(n) = 2n + 4$ ($n > 1$)。

考虑以下 R 和 M 地结合



M 可以以时间复杂性 $tc_M(n) = (n^2 + 3n + 4)/2$ 接收 Q 。对字符串 $x^n y^n z^n$ 而言, 如果 n 是偶数那么复合机器的性能最差; 对字符串 $x^{n+1} y^n z^n$ 而言, 如果 n 是奇数那么复合机器的性能最差。 L 成员资格问题的结果方案的复杂性是

$$tc_R(n) + tc_M(tc_R(n)) = 2n + 2 + (n^2 + 3n + 4)/2,$$

这个复杂性在定理 15.6.2 的证明给出的上界内。 \square

问题归约为我们比较两个问题的相对难度奠定了基础。如果两个问题的解的时间复杂性仅有多项式的差别, 那么我们认为他们的难度相同。也许应该指出, 与 $\Theta(n^1)$ 算法相比我们更倾向于 $\Theta(n^2)$ 算法, 并且人们花费了相当多的时间和精力来降低许多算法的复杂性。这是真实的 (也是值得我们努力的), 但是我们的重点是区分可解问题和难解问题。从这个角度看, 算法时间复杂性的多项式差别并不重要。

如果 L 可以在多项式时间归约到 Q , 那么 Q 可能被认为至少是跟 L 一样难的问题。自动化地寻找 Q 的解需要 L 的解; 通过顺序地执行归约得到解, 随后是 Q 的解。此外, 归约和 Q 的解的结合的时间复杂性说明, 如果 Q 是可解的, 那么 L 也是可解的。归约和语言的相对难度之间的关系可以扩展到语言类。

定义 15.6.3 设 \mathcal{C} 是一个语言类。一个语言 Q 难于类 \mathcal{C} , 如果 \mathcal{C} 中的每个语言都可以在多项式时间内归约到 Q 。

如果 Q 难于类 \mathcal{C} 并且在多项式时间内是有解的, 那么类 \mathcal{C} 中的每一个问题都在多项式时间内有解, 且 $\mathcal{C} \subseteq \mathcal{P}$ 。

15.7 $\mathcal{P} = \mathcal{NP}$?

一个可以被确定型多道或多带机器在多项式时间接收的语言在 \mathcal{P} 内。从这些选择中构造一个等价的标准图灵机能够保持多项式时间复杂性。8.7 节介绍了从非确定型机器的转换到等价确定型机器

的构造技术。不幸的是,根据定理 15.1.2,这个构造并不能保持多项式时间复杂性。

[479]

哈密尔顿回路问题的两个解法鲜明地说明了确定型和非确定型计算的不同。为了找到答案,确定型解法在试图发现周游中生成节点的序列。在最坏情况下,这个过程需要检查节点的所有可能包含图的周游的序列。非确定型机器通过“猜想”一个序列并判定其是否能构成一个周游来避免这个过程。 $P = NP$ 问题的哲学解释是,为问题构造一个解在本质上是否比检查一个可能是否满足了问题的条件更加困难。由于现在已知的额外困难,确定型解跨越非确定型解涉及很大范围的重要问题,一般大家普遍相信 $P \neq NP$ 。然而, $P = NP$ 问题是一个精确形式化的数学问题,并且只有当两个类的等价 P 包含在 NP 的结论被证明后才能解决。

判定是否 $P = NP$ 的一个方法是在独立的基础上检查每一个语言或者判定性问题的属性。例如,人们投入相当多的精力试图提出一个确定型多项式算法来解决哈密尔顿回路问题。事实上,找到这样一个解法仅能解决针对某种语言的问题。我们需要的是一个对于所有 NP 语言而言能够一次解决确定型多项式可解性问题的通用方法。对 NP 类而言,语言表示的困难使得我们可以把 NP 类中所有问题的多项式时间有解问题转换为一个问题的多项式时间有解问题。

定义 15.7.1 一个语言 Q 是 **NP 难的** (NP -hard),如果对于所有 $L \in NP$, L 可以在多项式时间内归约到 Q 。 NP 中 **NP 难的语言** 也叫做 **NP 完全的** (NP -complete)。

我们可以把一个 NP 完全的语言看作是 NP 类中的一个通用的语言。接收 NP 完全语言的多项式时间机器的发现可以用于构造在确定多项式时间内接收 NP 类中所有语言的机器。也就是说,这给出了 $P = NP$ 问题的肯定答案。

定理 15.7.2 如果存在一个在 P 中的 NP 完全语言,那么 $P = NP$ 。

证明: 假设 Q 是一个可以被确定型图灵机在多项式时间接收的 NP 完全语言。设 L 是 NP 中的任意语言。因为 Q 是 NP 难的,所以存在从 L 到 Q 的多项式时间归约。根据定理 15.6.2, L 也在 P 内。 ■

由于图灵机可计算性的概念和符号所提供的精确性, NP 完全性的定义使用了递归语言和图灵机计算函数的术语。递归语言和有解判定问题之间的二元性允许我们描述 NP 难和 NP 完全判定问题的说法。在判定性问题的上下文中重新检查这些定义是值得的。

使用图灵机计算函数的语言的归约性是第 11 章中提出的判定的问题归约概念的形式化。一个判定问题是 NP 难或 NP 完全的,无论解决这个问题的机器接收的语言是什么。使用了 NP 类问题到一个 NP 难问题的 P 通用的归约,我们可以通过结合归约和解决 P 的机器获得任何一个 NP 问题的解。

[480]

从语言或判定问题的角度看,不管我们是否接近了 NP 完全性,显然这都是一类重要问题。不幸的是,我们还没有说明这样一个一般问题的存在。尽管这需要大量的工作,这个遗漏将在下一部分弥补。

15.8 可满足性问题

可满足性问题与命题逻辑中公式的真值相关,它是用于说明 NP 完全的第一个判定问题。公式的真值可以通过公式中的基本命题获得。可满足性问题的目标是判定是否存在一个使命题为真的命题真值赋值。在说明可满足性问题是 NP 完全的以前,我们将简要地回顾一下命题逻辑的基础。

一个布尔变量 (boolean variable) 是一个值为 0 或者 1 的变量。布尔变量可以看作是命题,是命题逻辑的基本对象。变量的值描述了命题的真或假。当布尔变量被赋值为 1 时命题 x 为真。值 0 指派了一个假命题。一个真值赋值 (truth assignment) 是对每个布尔变量赋值为 0 或者 1 的函数。

命题连接词 \wedge (与)、 \vee (或) 和 \neg (否) 用于从布尔变量集构造命题,这些命题称为合式公式 (well-formed formula)。我们将使用符号 x, y, z 表示布尔变量,使用 u, v, w 表示合式公式。

定义 15.8.1 设 V 是布尔变量的集合。

- i) 如果 $x \in V$, 那么 x 是合式公式。
- ii) 如果 u, v 是合式公式, 那么 $(u), (\neg u), (u \wedge v)$ 和 $(u \vee v)$ 是合式公式。
- iii) 一个表达式是 V 上的合式公式, 仅当它可以从布尔变量集获得且有限次地应用了步骤 (ii)。

中的操作。

表达式 $((\neg(x \vee y)) \wedge z)$ 、 $((x \wedge y) \vee z) \vee \neg(x)$ 和 $((\neg x) \vee y) \wedge (x \vee z)$ 是布尔变量 x 、 y 和 z 上的合式公式。我们可以通过定义逻辑操作符的优选关系来减少合式公式中圆括号的数量。否可以看作是最紧密的操作符，随后是与操作符和或操作符。此外，与和或操作符的关联允许省略操作符序列中的圆括号。我们可以通过这些规则来把写上面的公式为 $\neg(x \vee y) \wedge z$ 、 $x \wedge y \vee z \vee \neg x$ 和 $(\neg x \vee y) \wedge (x \vee z)$ 。

481

变量的真值可以通过直接的真值赋值得到。逻辑操作的标准解释可用于把真值从变量扩展到合式公式。我们可以根据下表中定义的 u 和 v 的值的定义规则或得公式 $\neg u$ 、 $u \wedge v$ 和 $u \vee v$ 的真值。

u	$\neg u$	u	v	$u \wedge v$	u	v	$u \vee v$
0	1	0	0	0	0	0	0
1	0	0	1	0	0	1	1
		1	0	0	1	0	1
		1	1	1	1	1	1

如果变量的值可以使得 u 的值为 1，那么公式 u 通过真值赋值被满足。如果两个合式公式可以被同样的真值赋值满足，那么它们是等价的。

一个语句 (clause) 是一个变量或者其否定形式的析取构成的合式公式。非否定的变量叫做正文字 (positive literal)，否定变量叫做负文字 (negative literal)。使用这种术语，那么一个语句是文字的析取 (disjunction of literal)。公式 $x \vee \neg y$ 、 $\neg x \vee z \vee \neg y$ 和 $x \vee z \vee \neg x$ 是布尔变量集 $\{x, y, z\}$ 上的语句。一个公式是合取范式 (conjunctive normal form)，如果其形式为：

$$u_1 \wedge u_2 \wedge \cdots \wedge u_n,$$

其中每个 u_i 是一个语句。命题逻辑的一个经典定理是每一个合式公式都可以转换为等价的合取范式形式。

如前所述，可满足性问题是判定是否一个合取范式形式的公式可以被某些真值赋值满足。以下从变量 $\{x, y, z\}$ 上构造的公式

$$u = (x \vee y) \wedge (\neg y \vee z)$$

$$v = (x \vee \neg y \vee \neg z) \wedge (x \vee z) \wedge (\neg x \vee \neg y)$$

可以被如下的真值赋值满足。

其中， u 中的第一个语句可以被 x 满足，第二个语句可以被 $\neg y$ 满足。 v 的第一个语句可以被所有三个变量满足，第二个语句可以被 x 满足，第三个语句可以被 $\neg y$ 满足。公式

t	
x	1
y	0
z	0

$$w = \neg x \wedge (x \vee y) \wedge (\neg y \vee x)$$

482

不能被 t 满足。此外，我们不难看出 w 不能被任何真值赋值满足。

我们可以通过检查每一个真值赋值来获取可满足性问题的确定型解。可能的真值赋值的数量是 2^n ，其中 n 是布尔变量的数量。这个策略的应用在本质上是一个为公式构造完整的真值表的系统方法。显然，这种穷尽方式的复杂性是指数级的。然而，检查特定真值赋值的方面的扩展工作随变量数量和公式长度多项式地增长。这个发现为设计能够解决可满足性问题的多项式时间非确定型机器提供了思路。

定理 15.8.2 可满足性问题的在 \mathcal{NP} 中。

证明：我们首先为布尔变量集 x_1, \dots, x_n 上的合式公式提供一种表示法。我们把每一个变量编码为其下标的二进制形式。我们给正文字编码的最后添加 #1，在负文字编码的最后添加 #0。

文字	编码
x_i	$i\#1$
$\neg x_i$	$i\#0$

变量编码的数量描述了满足文字的布尔值。

我们通过连接文字和表示合取及析取关系的符号对合式公式进行编码。合取公式形式的公式

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$$

将被编码为

$$1\#1\vee 10\#0\wedge 1\#0\vee 11\#1.$$

最后, 机器的输入是由公式中变量的编码、##以及公式本身的编码组成的。上面公式的输入字符串如下所示:

$$\begin{array}{|c|c|} \hline 1\#10\#11\##1\#1\vee 10\#0\wedge 1\#0\vee 11\#1 \\ \hline \text{变量} \quad \quad \quad \text{公式} \\ \hline \end{array}$$

[483] 一个可满足性问题的实例的表示是一个字母表 $\Sigma = \{0, 1, \wedge, \vee, \#\}$ 上的字符串。语言 L_{SAT} 由所有 Σ 上的所有表示可满足的合取范式公式的字符串组成。

一个可以解决可满足性问题的双带非确定型图灵机 M 的描述如下: M 使用猜想和检查策略; 猜想非确定地生成一个真值赋值。表示公式 $(x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_4)$ 的输入字符串初始化计算相应的配置, 以便说明机器的活动。带的初始化配置包括带 1 上的公式表示和带 2 的空白:

带 2 BB

带 1 B1#10#11##1#1 \vee 10#0 \wedge 1#0 \vee 11#1B

1. 如果输入不是以上要求的形式, 那么计算停机并且拒绝该字符串。

2. 带 1 上第一个变量的被拷贝到带 2 上, 然后打印#并且非确定的写入 0 或 1。如果该变量不是最后一个变量, 那么写##, 并且该过程为下一个变量重复执行。非确定型地为每一个变量选择值定义了一个真值赋值 t 。将该值赋给变量 x_i 表示为 $t(x_i)$ 。使用这种表示后, 带有以下的形式:

带 2 B1# $t(x_1)$ ##10# $t(x_2)$ ##11# $t(x_3)$ B

带 1 B1#10#11##1#1 \vee 10#0 \wedge 1#0 \vee 11#1B

带 2 的带头被重新布置到最左端的位置。带 1 的带头移过##到新的位置以便读入公式的第一个变量。

真值赋值的生成是 M 非确定性的唯一实例。计算的后续部分将检查是否公式被非确定地选择的真值赋值所满足。

3. 假设带 1 上变量 x_i 的编码被扫描。那么 x_i 的编码在带 2 上可以被找到。机器后续的活动是由 $t(x_i)$ 与带 2 上那个紧随带 1 上 x_i 的布尔值的比较结果决定的。

4. 如果值不匹配, 那么当前的文字不能被真值赋值满足。如果文字后续字符是 B 或者 \wedge , 那么当前语句中的每个文字都已经被检查并且以失败告终。当这种情况发生时, 真值赋值不能满足公式并且计算停机在不接收状态。如果读入的是 \wedge , 那么带头被布置以便检查语句中的下一个文字 (步骤 3)。

5. 如果值不匹配, 那么文字和当前的语句不能被真值赋值满足。带 1 的带头向右移动至下一个 \wedge 或者 B 。如果遇到 B , 那么计算停止并接收输入, 否则, 返回步骤 3 处理下一个语句。

步骤 3 的匹配过程决定了计算的时间复杂性的增长速度。在最坏的情况下, 匹配过程需要比较带 1 上的变量和带 2 上的每一个变量, 以便能够发现匹配。这可以在 $O(k \cdot n^2)$ 时间内完成, 其中 n 是变量的数量, k 是输入中文字的数量。

[484]

现在我们必须说明 L_{SAT} 是 NP 难的, 也就是说, NP 中的每一种语言都可以在多项式时间可归约到 L_{SAT} 的。首先, 这看上去像是一个不可能的任务。NP 中有无穷多种语言, 它们似乎很少有共性, 它们甚至没有被限制用同样的字母表。NP 语言的唯一共同特性是它们都能被多项式时间边界的非确定型图灵机接收。不幸地是, 这远远不够。除了考虑到语言, 证明还将揭示可以接收语言的机器的属性。在这种情况下, 我们可以用一个通用的过程把 NP 中的任意一种语言归约到 L_{SAT} 。

定理 15.8.3 (库克定理) 可满足性问题是 NP 难的。

证明: 设 L 是可以被非确定型图灵机 M 接收的, 且 M 的计算边界是多项式 p 。可满足性问题的 L 的归约可以把 M 计算的输入 u 转换为合取范式公式 $f(u)$, 从而使得 $u \in L(M)$ 当且仅当 $f(u)$ 是可满足的。接下来我们将说明构造 $f(u)$ 所需要的时间仅随 u 的长度多项式地增长。

在不损失一般性的情况下, 我们假设所有 M 的计算在一个或两个状态内停机。所有接收计算在

状态 q_A 停机, 所有不接收计算在状态 q_R 停机。此外, 我们假设其他状态不存在转换。通过添加每一个接收配置到 q_A 的转换和每一个拒绝配置到 q_R 的转换, 任意一个机器都可以转换为与之等价并满足以上条件的机器。这种变化在原机器的每一个计算上都增加了一个转换。从计算到合式公式的转换, 假设了所有输入长度为 n 的计算包括 $p(n)$ 中配置。如果有必要的话, 为了保证正确数量的配置出现, 终止配置将被重复。

M 的状态、停机状态和字母表表示为:

$$\begin{aligned} Q &= \{q_0, q_1, \dots, q_m\} \\ \Gamma &= \{B = a_0, a_1, \dots, a_s, a_{s+1}, \dots, a_t\} \\ \Sigma &= \{a_{s+1}, a_{s+2}, \dots, a_t\} \\ F &= \{q_m\} \end{aligned}$$

我们假设空白是编号为 0 的带字符。输入字母表包括了编号从 $s+1$ 到 t 的带字母表的元素。唯一的接收状态是 q_m , 拒绝状态是 q_{m-1} 。

设 $u \in \Sigma^*$ 是长度为 n 的字符串。我们的目标是定义一个公式 $f(u)$, 这个公式能够对 M 计算的输入 u 编码。 $f(u)$ 的长度依赖于输入长度为 n 时 M 计算的最大转换次数 $p(n)$ 。我们设计这样一个编码, 使得存在一个真值赋值满足 $f(u)$ 当且仅当 $u \in L(M)$ 。公式是根据三类变量构造的, 每一类都用于表示某种机器配置的属性。

485

变量	解释 (满足时)
$Q_{i,k}$	$0 \leq i \leq m$ $0 \leq k \leq p(n)$ 时间为 k 时, M 在状态 q_i
$P_{j,k}$	$0 \leq j \leq p(n)$ $0 \leq k \leq p(n)$ 时间为 k 时, M 正在扫描位置 j
$S_{i,r,k}$	$0 \leq j \leq p(n)$ $0 \leq r \leq t$ $0 \leq k \leq p(n)$ 时间为 k 时, 带上的 j 位置包含符号 a_r

变量集合 V 是上表中定义的三个集合的并集。 M 的计算定义了 V 的一个真值赋值。例如, 如果初始时带位置 3 包含符号 a_i , 那么 $S_{i,3,0}$ 为真。同时对所有 $i \neq j$, $S_{j,3,0}$ 必须为假。通过这种方式获得的真值赋值描述了状态、带头位置和时间 k ($0 \leq k \leq p(n)$) 时带上的字符。这是计算生成的配置序列的详细信息。

对 V 中变量的任意一个真值赋值都不必与 M 的一个计算对应。给 $P_{0,0}$ 和 $P_{1,0}$ 同时赋值为 1 表示时间为 0 时带头在两个截然不同的位置。类似的, 一个真值赋值可能描述了机器在某个给定的时间处于几个状态或者在某个位置指派了多个符号。

公式 $f(u)$ 应该说明为了保证变量的解释与通过计算中真值赋值得到的是相同的所需要的约束条件。我们根据 M 输入字符串 u 和转换定义了八个公式集合。这八个公式集合中的七个是以语句的形式给出的。我们可以以图灵机配置和计算的形式来简要地描述他们的解释, 语句可以通过这些描述获得。下面这两种记法

$$\bigwedge_{i=1}^k v_i \quad \bigvee_{i=1}^k v_i$$

分别表示了文字 v_1, v_2, \dots, v_k 的合取和析取。

一个满足下表 (i) 定义的语句集合的真值赋值表示机器每次只处于一个状态。满足第一个合取保证了至少一个 $Q_{i,k}$ 中的变量为真。否定对表示两个状态不可能同时满足。最简单的这种情况是, 使用蕴含 $A \Rightarrow B$ 的等价析取形式 $\neg A \vee B$ 把 $\neg Q_{i,k} \vee \neg Q_{i',k}$ 转换为其蕴含的形式, 我们把 $\neg Q_{i,k} \vee \neg Q_{i',k}$ 写成蕴含 $Q_{i,k} \Rightarrow \neg Q_{i',k}$, 后者可以被解释为一个断言, 这个断言检查对于任意的 $i' \neq i$, 机器在时间 k 时是否处于状态 q_i 和 $q_{i'}$ 。

486

语 句	条 件	解 释 (满足时)
i) 状态		
$\bigvee_{i=0}^m Q_{i,k}$	$0 \leq k \leq p(n)$	每次时间为 k 时 M 至少在一个状态
$\neg Q_{i,k} \vee \neg Q_{i',k}$	$0 \leq i < i' \leq m$ $0 \leq k \leq p(n)$	M 至少在一个状态 (现在同时共有两种不同的状态)
ii) 带头位置		
$\bigvee_{j=0}^{p(n)} P_{j,k}$	$0 \leq k \leq p(n)$	每次时间为 k 时带头至少在一个位置
$\neg P_{j,k} \vee \neg P_{j',k}$	$0 \leq j < j' \leq p(n)$ $0 \leq k \leq p(n)$	最多一个位置
iii) 带上符号		
$\bigvee_{r=0}^t S_{j,r,k}$	$0 \leq j \leq p(n)$ $0 \leq k \leq p(n)$	对于每次时间 k 和位置 j , 位置 j 包括至少一个符号
$\neg S_{j,r,k} \vee \neg S_{j,r',k}$	$0 \leq j \leq p(n)$ $0 \leq r < r' \leq t$ $0 \leq k \leq p(n)$	最多一个符号
iv) 字符串 $u = a_{r_1}, a_{r_2}, \dots, a_{r_n}$ 的初始条件		
$Q_{0,0}$		
$P_{0,0}$		计算开始读最左边的空白
$S_{0,0,0}$		
$S_{1,r_1,0}$		时间为 0 时, 字符串 u 在输入位置
$S_{2,r_2,0}$		
\vdots		
$S_{n,r_n,0}$		
$S_{n+1,0,0}$		
\vdots		时间为 0 时, 带的其余部分是空白
$S_{p(n),0,0}$		
\vdots		
v) 接受条件		
$Q_{m,p(n)}$		计算的停机状态是 q_m

因为输入为 n 时 M 的计算不能访问带上 $p(n)$ 以外的位置, 因此一个机器配置完全是通过状态、带头位置和带上初始位置 $p(n)$ 的内容来定义的。一个满足 (i), (ii) 和 (iii) 语句的真值赋值给 0 和 $p(n)$ 之间的每个时间都定义了一种机器配置。语句 (i) 和 (ii) 的合取说明机器每次处于一个状态且扫描带上的一个位置。(iii) 中的语句保证了带是定义良好的; 也就是说, 带在每个位置定义一个字符, 计算过程中这个字符会被引用。

一个计算不包含不相关配置的序列, 而是那些在一个转换后能与其前驱有所区分的配置序列。我们必须添加那些满足条件描述了时间 0 时的配置并且连接到连续的配置的语句。最初, 机器处于状态 q_0 , 带头扫描最左边的位置, 输入位于带上 1 到 n 位置, 并且带上剩余的部分都是空白。(iv) 中 $p(n)+2$ 语句的满足条件保证了时间为 0 时正确的机器配置。

通过实施一个转换, 每一个后续的配置都必须从其后继得到。假设机器处于状态 q_i , 在时间 k 时正在扫描位置 j 处的字符 a_r 。我们介绍最后三类公式, 根据机器 M 的转换和时间 k 时定义的变量来生成时间 $k+1$ 时允许的配置。

带上一个转换的影响是重写带头扫描的位置。考虑到位置 $P_{j,k}$ 可能的异常, 时间 $k+1$ 时每一个带上的位置都包含了与时间 k 时相同的字符。语句必须被添加到公式中以保证带上其余的部分不被转换影响。

语 句	条 件	解 释 (满足时)
(vi) 带一致性		
$\neg S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$	$0 \leq j \leq p(n)$ $0 \leq r \leq t$ $0 \leq k \leq p(n)$	不在带头位置的符号不会改变

如果带上一个位置发生改变而且不是被扫描的带头位置, 那么这个语句不能被满足. 这也就是

$$\neg S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$$

等价于

$$\neg P_{j,k} \Rightarrow (S_{j,r,k} \Rightarrow S_{j,r,k+1})$$

这清晰地说明了如果时间 k 时带头不在位置 j , 那么时间 $k+1$ 时位置 j 的符号跟时间 k 时位置 j 的符号相同。

488

现在假设在给定的一个时间 k , 机器处于状态 q_i 并正在扫描位置 j 处的字符 a_r . 通过把布尔变量 $Q_{i,k}$ 、 $P_{j,k}$ 和 $S_{j,r,k}$ 赋值为 1 可以指派一个配置的这些特征. 当 $Q_{i,k+1}$ 为真时语句 (a) 可以被满足. 在计算时, 这意味着 M 在时间 $k+1$ 时进入了状态 q_i . 类似地, 时间 $k+1$ 时位置 j 处的符号和带头位置由语句 (b) 和语句 (c) 描述. 其中 $n(L) = -1$ 且 $n(R) = 1$.

$$\text{a) } \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i',k+1}$$

$$\text{b) } \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r',k+1}$$

$$\text{c) } \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j+n(d),k+1},$$

语句 (a)、(b) 和 (c) 的合取能够被满足, 仅当时间 $k+1$ 时的配置是通过实施一个转换 $[q_i', a_r', d] \in \delta(q_i, a_r)$ 从时间 k 时的配置中得到的。

我们可以用转换的语句表示来构造一个公式, 这个公式的满足能够保证时间 $k+1$ 时配置中定义的变量是通过 M 实施一个转换从时间 k 时定义的配置中得到的. 除了状态 q_m 和 q_{m-1} , M 的约束保证了给每个状态和每个字符对都至少定义了一个转换。

合取范式公式

$$\begin{aligned} & (\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i',k+1}) && \text{(新状态)} \\ & \wedge (\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j+n(d),k+1}) && \text{(新带头位置)} \\ & \wedge (\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r',k+1}) && \text{(位置 } r \text{ 处的新符号)} \end{aligned}$$

是针对每个

$$\begin{aligned} & 0 \leq k \leq p(n) && \text{(时间)} \\ & 0 \leq i < m-1 && \text{(非停机状态)} \\ & 0 \leq j \leq p(n) && \text{(带头位置)} \\ & 0 \leq r \leq t && \text{(带符号)} \end{aligned}$$

构造的, 其中 $[q_i', a_r', d] \in \delta(q_i, a_r)$, 除非位置是 0 并且方向 L 是由一个转换描述的. 当一个转换的实施会导致带头穿过带的左边界时这种异常状况会发生. 以语句的形式看, 这表示后续配置包括拒绝状态 q_{m-1} . 这种特殊情况由以下公式编码

$$\begin{aligned} & (\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee Q_{m-1,k+1}) && \text{(进入拒绝状态)} \\ & \wedge (\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee P_{0,k+1}) && \text{(同样的带头位置)} \\ & \wedge (\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee S_{0,r,k+1}) && \text{(同样的位置 } r \text{ 处的符号)} \end{aligned}$$

其中对于所有转换都有 $[q_i', a_r', L] \in \delta(q_i, a_r)$ 。

489

由于 M 是非确定型的, 所以存在一些能够用于给定配置的转换. 任何一个这种变化的结果都是计算中允许的后续配置. 设 $\text{trans}(i, j, r, k)$ 是一个表示状态 q_i 中的时间 k , 带头位置在 j , 带上符号 r 的所有合取范式的析取. 公式 $\text{trans}(i, j, r, k)$ 才能被满足, 仅当时间 $k+1$ 时配置编码的变量的值表示时间 k 时配置编码的变量的一个合法后续。

公 式	解 释 (满足时)
vii) 生成后续配置转换 $trans(i, j, r, k)$	通过实施一个转换, 配置 $(k+1)$ 紧随配置 k 之后.

当机器处于停机状态 q_m 或 q_{m-1} 时, 公式 $trans(i, j, r, k)$ 不描述将要发生的动作. 在这种情况下, 后续的配置与其前驱相同.

公 式	解 释 (满足时)
viii) 停机计算	
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i,k+1}$	(同样的状态)
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j,k+1}$	(同样的带头位置)
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r,k+1}$	(位置 r 处同样的符号)

这些语句的构造要求所有 i, j, r 和 k 在适当的范围内且 $i = q_{m-1}, q_m$.

设 $f(u)$ 是(i)和(viii)构造的语句的合取公式. 当 $f(u)$ 被一个 V 上的真值赋值满足时, 这些变量就定义了 M 的接收字符串 u 的计算. 条件(iv)的语句说明时间 0 时的配置是输入为 u 的 M 的计算的初始配置. 每一个后续的配置都通过其前驱实施一个转换的结果得到. 由于(v)的满足条件表示最终的配置包括接收状态 q_m , 因此机器 M 接收字符串 u .

我们可以利用引理 15.8.4 介绍的技术将每个 $trans(i, j, r, k)$ 公式转换为合取范式, 从而可以从 $f(u)$ 得到合取范式公式 $f(u)$. 下面将说明从字符串 $u \in \Sigma^*$ 到 $f(u)$ 的转换可以在多项式时间内完成.

从 u 到 $f(u)$ 的转换包含构造语句和把 $trans$ 转换为合取范式. 语句的数量是下列内容的函数

i) 状态的数量 m 和带字符的数量 t ,

ii) 输入字符串 u 的长度 n , 以及

iii) M 计算长度的边界 $p(n)$.

[490]

m 和 t 的值可以通过图灵机 M 获得, 并且它们与输入字符串无关. 从下标的范围我们可以看到, 语句的数量在多项式 $p(n)$ 内. $f(u)$ 的建立可以通过转换成合取范式来完成, 根据引理 15.8.4, 这是公式 $trans(i, j, r, k)$ 中语句数量的多项式.

我们已经知道合取范式公式可以在一定的步骤内完成, 步骤的数量随着输入字符串长度做多项式地增长. 我们真正需要的是作为解决可满足性问题的图灵机输入的代表. 任何合理的编码, 包括定理 15.8.2 的编码方式, 把高层表示转换为机器表示都只需要多项式时间. ■

上面的证明遗漏了将公式 $trans(i, j, r, k)$ 转换为合取范式的步骤. 下面的引理将说明任何合取范式公式的析取都可以在多项式时间内被转换为合取范式.

引理 15.8.4 设 $u = w_1 \vee w_2 \vee \cdots \vee w_n$ 是布尔变量集 V 上合取范式公式 w_1, w_2, \dots, w_n 的析取形式. 同时, 设 $V' = V \cup \{y_1, y_2, \dots, y_{n-1}\}$, 其中变量 y_i 不在 V 中. 公式 u 可以被转换为 V' 上的公式 u' , 使得

i) u' 是合取范式;

ii) u' 在 V' 上是可满足的, 当且仅当, u 在 V 上是可满足的; 并且

iii) 转换可以在 $O(m \cdot n^2)$ 时间内完成, 其中 m 是 w 中语句的数量.

证明: 两个合取范式公式的析取的转换已经给出. 这个技术可以被重复 $n-1$ 次以便转换 n 个公式的析取. 设 $u = w_1 \vee w_2$ 是两个合取范式公式的析取, 那么 w_1 和 w_2 可以写为

$$w_1 = \bigwedge_{j=1}^l \left(\bigvee_{k=1}^t v_{j,k} \right)$$

$$w_2 = \bigwedge_{i=1}^l \left(\bigvee_{k=1}^t p_{i,k} \right),$$

其中 r_j 是 w 中语句的数量, s_j 是 w_1 中第 j 个语句的文字数量, t_j 是 w_2 中第 j 个语句的文字数量。定义

$$u' = \bigwedge_{j=1}^r (y \vee \bigvee_{k=1}^{s_j} v_{j,k}) \wedge \bigwedge_{j=1}^t (\neg y \vee \bigvee_{k=1}^{t_j} p_{j,k}).$$

公式 u' 可以通过分派 y 到 w_1 中的每一个语句和分派 $\neg y$ 到 w_2 中的每一个字句得到。

现在我们证明无论 u 是什么, u' 都是可满足的。假设 w_1 可以被 V 上的真值赋值 t 满足。那么真实赋值 t' 能够满足 u' 。

$$t'(x) = \begin{cases} t(x) & \text{如果 } x \in V \\ 0 & \text{如果 } x = y \end{cases}$$

当 w_2 可被 t 满足时, 真实赋值 t' 可以通过扩展 t 设 $t'(y) = 1$ 得到。

反之, 假设 u' 能够被真值赋值 t' 满足, 那么 t' 对 V 的约束满足 u 。如果 $t'(y) = 0$, 那么 w_1 一定为真。另一方面, 如果 $t'(y) = 1$, 那么 w_2 为真。

下面的转换

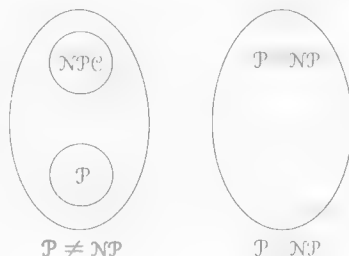
$$u = w_1 \vee w_2 \vee \cdots \vee w_n$$

只需要将以上过程迭代 $n-1$ 次。这种重复在 w_1 和 w_2 的每个语句中添加了 $n-1$ 个文字, 在 w_1 的每个语句中添加了 $n-2$ 个文字, 在 w_1 的每个语句中添加了 $n-3$ 个文字, 依此类推。转换所需要的步骤少于 $m \cdot n^2$, 此处 m 是公式 w_1, w_2, \dots, w_n 的语句数量。 ■

15.9 复杂类的关系

我们在本章的最后介绍两个图, 这两个图说明了前面介绍的复杂类之间可能存在的关系。包含了所有 NP 完全问题的类叫做 NPC, 可满足性问题的存在保证了这个类是非空的。

如果 $\mathcal{P} \neq \mathcal{NP}$, 那么 \mathcal{P} 和 NPC 都是非空的, 都是 \mathcal{NP} 的子集。大多数数学家和计算机界的科学家都相信这种假设是真的。另一种不太可能的情况是 \mathcal{P} 确实等于 \mathcal{NP} , 那么这两个集合将合并为一个集合。练习 17 要求你在这种可能的情况下确定 NP 完全问题的集合



[492]

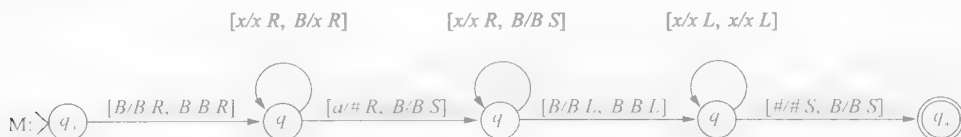
15.10 练习

1. 设 M 是图灵机



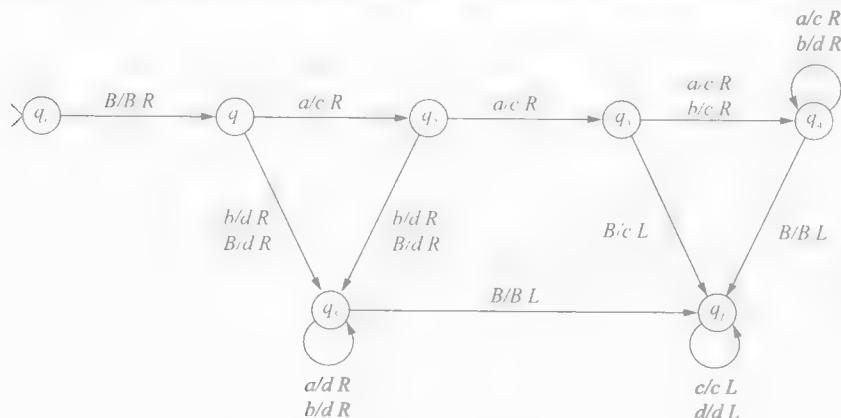
- 跟踪输入为 λ, a, aa 时 M 的所有计算。
- 描述输入为 a^n 时 M 的计算所要求的最大转换次数。
- 给出函数 tc_M 。

2. 设 M 是图灵机



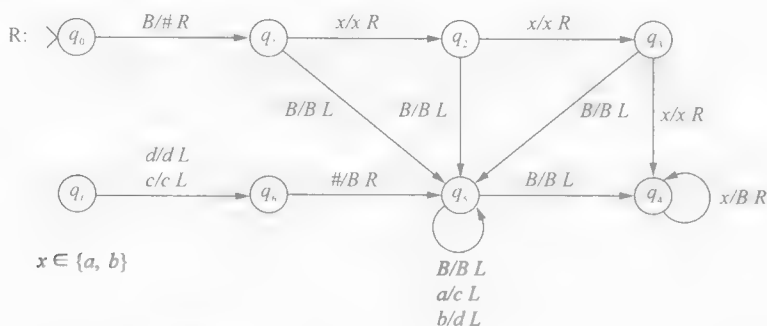
其中 x 表示 a 或者 b 。

- a) 跟踪输入为 $bbabb$ 时 M 的计算。
 - b) 给出 M 的语言的集合论定义。
 - c) 什么样的长度为 n 的字符串会导致最多次转换? 为什么?
 - d) 给出函数 tc_M 。
3. 说明类 \mathcal{P} 对于并、连接和补计算是闭包的。
4. 说明 \mathcal{NP} 类对于并、连接和克林星运算是闭包的。
5. 设 $L = (R(M)w \mid M \text{ 接收 } w \text{ 最多执行 } 2^{\text{length}(w)} \text{ 次转换})$
- a) 证明 L 不在 \mathcal{P} 内。提示: 利用 \mathcal{P} 在补运算上的闭包证明。如果 L 在 \mathcal{P} 内, 那么存在一个图灵机 M' 能够接收 M 的所有 $R(M)$ 表示, 并且 M 不在 $2^{\text{length}(R(M))}$ 次转换内接收它们自己的表示, 然后使用自引用可得到冲突。
 - b) 证明 L 不在 \mathcal{NP} 内。
6. 请设计一个双带图灵机, 使之能把一元数字转换为二进制数字。试确定该机器的时间复杂性。
7. 请设计一个双带图灵机, 使之能把二进制数字转换为一元数字。请解释为什么该转换不能在多项式时间内完成。
8. 设 P 是一个输入包含自然数的判定问题, M 是一个使用二进制表示法在多项式时间内能解决 P 的图灵机。请使用 M 设计一个机器, 使之能使用 3 为基数的自然数表示法解决 P 。请说明该解法也是多项式的。
9. 设 M 是一个非确定型图灵机, p 是一个多项式。假设 $L(M)$ 的每一个长度为 n 的字符串都能在至少一个 $p(n)$ 计算或者更少的转换内被接收。请注意没有关于不接收计算或其他接收计算的长度说明。证明 $L(M)$ 在 \mathcal{NP} 内。
10. 请构造一个确定型图灵机, 使之能在多项式时间内将语言 L 归约到 Q 。请使用大 O 表示法给出该机器计算下列归约的时间复杂性。
- a) $L = \{a^i b^j c^i \mid i \geq 0, j \geq 0\}$ $Q = \{a^i c^i \mid i \geq 0\}$
 - b) $L = \{a^i (bb)^i \mid i \geq 0\}$ $Q = \{a^i c^i \mid i \geq 0\}$
 - c) $L = \{a^i b^i a^i \mid i \geq 0\}$ $Q = \{c^i d^i \mid i \geq 0\}$
11. 机器 R 能够执行一个多项式时间归约, 把语言 $L = aa(a \cup b)^*$ 归约为语言 $Q = ccc(c \cup d)^*$ 。



- a) 请分别跟踪输入字符串为 $aabb$ 和 $abbb$ 时 R 的计算。
- b) 什么样的长度为 n 的字符串可以导致 R 执行最多次转换? 为什么?
- c) 给出时间复杂性函数 $tc_R(n)$ 。

12. 机器 R



计算从 $\{a, b\}^*$ 到 $\{c, d\}^*$ 的函数。

a) 使用 \vdash 符号跟踪输入字符串为 $abba$ 时 R 的计算。

b) 什么样的长度为 n 的字符串会导致 R 执行最多次转换？为什么？

c) 给出 $\tau_R(n)$ 。请用公式并解释为什么你给出的公式是正确的。

d) 机器 R 是否能把语言 $L = abb(a \cup b)^*$ 归约到语言 $Q = (c \cup d)cdd^*$ ？如果能，证明 R 计算的函数是一个归约。如果不能，请给出一个字符串说明这种映射不是归约。

13. 请给出满足下列每个公式的真值赋值：

a) $(x \vee y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y \vee \neg z)$

b) $(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

c) $(x \vee y) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (\neg y \vee \neg z)$

14. 请说明公式 $(x \vee \neg y) \wedge (\neg x \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (y \vee z)$ 是不可满足的。15. 请在 $\{x, y, z\}$ 上构造四个语句，使得它们中任意三个的合取是可满足的，而所有四个的合取是不可满足的。16. 证明公式 u' 是可满足的，当且仅当 u 是可满足的。

a) $u = v, u' = (v \vee y \vee z) \wedge (v \vee \neg y \vee z) \wedge (v \vee y \vee \neg z) \wedge (v \vee \neg y \vee \neg z)$

b) $u = v \vee w, u' = (v \vee w \vee y) \wedge (v \vee w \vee \neg y)$

17. 假设 $\mathcal{P} = \mathcal{NP}$

a) 设 L 是一个 \mathcal{NP} 中的一个语言， $L \neq \emptyset$ 且 $\bar{L} \neq \emptyset$ 。请证明 L 是 NP 完全的。

b) 为什么 \mathcal{NPC} 是 \mathcal{NP} 的真子集？

495

参考文献注释

Cobham [1964] 介绍了 \mathcal{P} 家族。Edmonds [1965] 首先研究了 \mathcal{NP} 问题。Cook [1971] 奠定了 NP 完全性理论的基础。这项工作包括了可满足性问题是 NP 完全的证明。Garey 和 Johnson [1979] 的经典著作给出了关于复杂性分析和 NP 完全性的精彩介绍。此外，它还是 20 世纪 70 年代关于 NP 完全问题的百科全书。

496

第 16 章 NP-完全问题

我们通过关联图灵机计算和合式范式公式说明了可满足性问题是 NP-完全的。如果每一个 NP-完全性证明都需要设计这种灵巧的转换，那么很多 NP-完全的问题都需要证明。幸运的是，问题归约为我们提供了一种可选的、通常也是简单的证明问题是 NP-完全的方法。将一个 NP-完全问题归约为另外一个 NP 中的问题就证明了后一个问题为 NP-完全问题。使用这个技术，我们可以从一组规则中获得问题的 NP-完全性结果。此外，我们还扩展 NP-完全性的表示来最优化问题。

如果一个问题被证明是 NP-完全的，那么试图发现多项式时间解几乎是不可能成功的。相比寻找解决问题的高效算法，遇到 NP-完全问题时采用不同的策略更加可行。一个可选的策略就是设计一个平均时间复杂性好、但是某些情况会表现出指数级性能的算法。在最优化问题时，接收一个近似最优解可能会减小问题的时间复杂性。在本章的最后，我们将介绍遇到 NP-完全问题时可以考虑的可选策略。

16.1 归约和 NP-完全问题

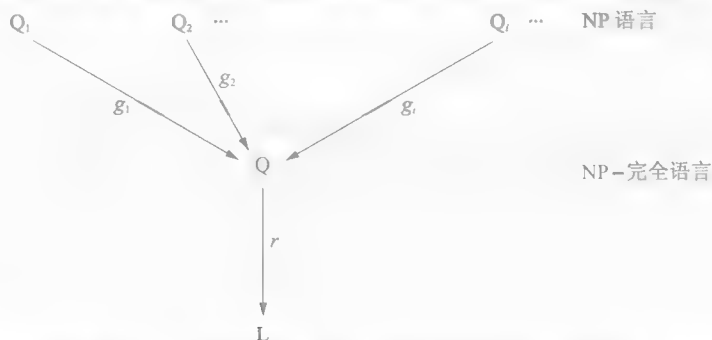
[497]

一个语言是 NP-完全的要具备两个条件：问题必须在 NP 内且必须是 NP-难的。满足前一个条件的最一般方式是简单地设计一个能在多项式时间内解决问题的非确定型算法。为了证明语言 L 是 NP-难的，证明 NP 中的每种语言都能在多项式时间归约到 L 是必要的。在做这种归约时，我们不是直接的生成到 L 的归约，而是把另外一个已知的 NP-完全问题作为中间步骤。

定理 16.1.1 设 Q 是一个 NP-完全语言。如果 Q 可以在多项式时间归约到 L，那么 L 是 NP-难的。

证明：设 r 是在多项式时间内将 Q 归约到 L 的计算函数，设 Q_i 是 NP 中的任意一种语言。由于 Q 是 NP-完全的，因此存在一个能够将 Q_i 归约到 Q 的计算函数 g_i 。复合函数 $r \circ g_i$ 是 Q_i 到 L 的归约。归约的多项式时间边界可以通过 r 和 g_i 的边界获得。 ■

通过归约证明一种语言是 NP-难的复合可以表示为如下图所示的包含两个步骤的过程：



第一个层次表示 NP 中任意语言 Q_i 通过函数 g_i 到 Q 的多项式时间可归约性。从 Q_i 到 L 的箭头表示 NP 中任意语言到 L 的可归约性。如果计算 g_i 和 r 的机器的时间复杂性分别是 $O(n^i)$ 和 $O(n^j)$ ，那么复合函数 $r \circ g_i$ 的时间复杂性就是 $O(n^j)$ ，并且从 Q_i 到 L 的归约可以在多项式时间内完成。在下面三个部分中，我们将使用定理 16.1.1 介绍几个额外的 NP-完全问题。

16.2 三元可满足性问题

三元可满足性问题是 NP-完全的可满足性问题的子问题。一个公式叫做三元合取范式 (3-conjunctive normal form), 如果它是一个合取范式且每个语句正好包含三个文字。三元可满足性问题的目标是确定是否存在一个可满足的三元合取范式公式。

498

使用第11章介绍的归约描述, 证明三元可满足性问题是 NP-难的所需的条件如下:

归约	输入	条件
可满足性	合取范式公式 u	u 是可满足的
到	\downarrow	当且仅当
三元可满足性	三元合取范式公式 u'	u' 是可满足的

也就是说, 归约必须是从任意一个合取范式公式到某个满足上述条件的三元合取范式公式。此外, u' 的构造必须能够在 u 长度的多项式时间内完成。

定理 16.2.1 三元可满足性问题是 NP-完全的。

证明: 显然, 三元可满足性问题的在 NP 内。对任意的合取范式公式而言, 解决可满足性问题的机器对其子集三元合取范式公式也能解决可满足性问题。

我们必须说明每一个合取范式公式 $u = w_1 \vee w_2 \vee \cdots \vee w_m$ 可以被转换为一个三元合取范式公式 u' , 从而使得 u 是可满足的当且仅当 u' 是可满足的。 u' 的构造可以通过将 u 中每个语句 w_i 独立地转换为一个三元合取范式公式 w'_i 来完成。这个转换必须设计成能够保证 w'_i 是可满足的当且仅当存在一个真值赋值满足原来的语句 w_i 。我们假设添加到语句转换中的变量不在 w_i 以外的其他地方出现。

如果 w_i 有三个文字, 那么不需要转换即有 $w_i = w'_i$ 。设 w 是 u 的语句且不包含三个文字。从 w 到三元合取范式公式的转换基于 w 中文字的数量:

长度 1: $w = v_1$

$$w' = (v_1 \vee y \vee z) \wedge (v_1 \vee \neg y \vee z) \wedge (v_1 \vee y \vee \neg z) \wedge (v_1 \vee \neg y \vee \neg z)$$

长度 2: $w = v_1 \vee v_2$

$$w' = (v_1 \vee v_2 \vee y) \wedge (v_1 \vee v_2 \vee \neg y)$$

长度 $n > 3$: $w = v_1 \vee v_2 \vee \cdots \vee v_n$

$$w' = (v_1 \vee v_2 \vee y_1) \wedge (v_3 \vee \neg y_1 \vee y_2) \wedge \cdots \wedge (v_j \vee \neg y_{j-2} \vee y_{j-1}) \wedge \cdots \wedge (v_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (v_{n-1} \vee v_n \vee \neg y_{n-3})$$

长度为 1 和长度 2 的语句的可满足性之间的关系证明以及转换留作习题。设 V 是语句 $w = v_1 \vee v_2 \vee \cdots \vee v_n$ 中的变量, 设 t 是一个满足 w 的真值赋值。因为 w 可以被 t 满足, 所以至少存在一个能被 t 满足的文字。设 v_j 是第一个这样的文字。那么下列真值赋值能够满足 w' 。

499

$$t'(x) = \begin{cases} t(x) & \text{如果 } x \in V \\ 1 & \text{如果 } x = y_1, \dots, y_{j-2} \\ 0 & \text{如果 } x = y_{j-1}, \dots, y_{n-3} \end{cases}$$

第一组 $j-2$ 个语句可以被文字 y_1, \dots, y_{j-2} 满足。最后一组 $n-j+1$ 个语句可以被 $\neg y_{j-1}, \dots, \neg y_n$ 满足。剩余的语句 $v_j \vee \neg y_{j-2} \vee y_{j-1}$ 可以被 v_j 满足。

反之, 设 t' 是满足 w' 的真值赋值。我们可以通过限制 t' 到 V 满足 w 来获得真值赋值 t 。证明的方法是反证法。假设 t 不满足 w , 那么没有语句 v_j ($1 \leq j \leq n$) 可以被 t 满足。因为 w' 的第一个语句值为 1, 因此 $t'(y_1) = 1$ 。现在, 由于第二个语句的值也为 1, 因此 $t'(y_2) = 1$ 。使用同样的推理, 我们可以得到 $t'(y_k) = 1$ 对于所有满足 $1 \leq k \leq n-3$ 的 k 都成立。这表示 w' 最后一个语句的值为 0, 这显然是一个矛盾, 因为 t' 被假设满足 u' 。

显然, 每个语句到三元合取范式公式的转换都是语句中文字数量的多项式。这项工作要求三元合取范式公式的构造是转换独立语句工作的总和。因此, 构造是原公式中语句数量的多项式。

NP-完全问题的子问题并不自动地就是 NP-完全的。判定合取范式公式的语句是否正好包含两个

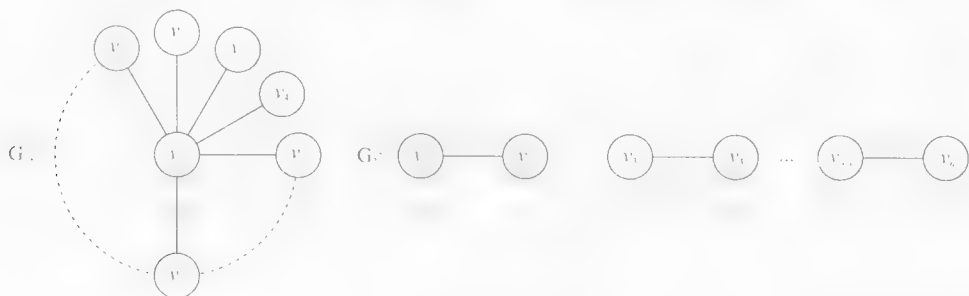
文字的三元可满足性问题,有多项式时间解(练习1)。因此三元可满足性问题不是 NP-完全的除非 $\mathcal{P} = \mathcal{NP}$ 。

16.3 三元可满足性的归约

以上我们给出的两个 NP-完全问题都关注于逻辑公式的满足情况。这一部分我们将扩展 NP-完全问题集合的范围,引入集合覆盖、图的路径、值得累积等相关的问题。三元合取范式公式的结构使得它们非常适用于设计其他领域问题的归约。在本章的后续部分,我们将使用问题实例的高层表示法来描述归约。

我们要考虑的第一个问题是顶点覆盖问题(vertex cover problem)。无向图 $G = (N, A)$ 的顶点覆盖是一个 N 的子集 VC ,使得对于 A 中的每条边 $|u, v|$, VC 至少包含 u 和 v 中的一个。我们可以这样描述顶点覆盖问题:给定无向图 G 和整数 k ,是否存在包含 k 个或少于 k 个顶点的 G 的顶点覆盖? 例 16.3.1 说明顶点覆盖的大小没有必要与图中节点数量或边的数量相关。

例 16.3.1 图 G_1 的边被一个顶点 v_1 覆盖。图 G_2 的最小顶点覆盖要求 $n/2$ 个顶点,每个顶点覆盖图中的一条边。



定理 16.3.1 顶点覆盖问题是 NP-完全的。 □

证明: 易知顶点覆盖问题在 \mathcal{NP} 中。非确定型解的策略由选择一个 k 个顶点的集合和判定他们是否覆盖了图中的边。我们通过将顶点覆盖问题归约到三元可满足性问题来说明该问题是 NP-难的。

归约	输入	条件
三元可满足性 到 顶点覆盖问题	三元合取范式公式 u ↓ 无向图 $G = (N, A)$, 整数 k	u 是可满足的 当且仅当 G 有一个大小为 k 的顶点覆盖

也就是说,对于任意三元合取范式公式 u ,我们必须构造一个图 G 使得 G 有一个大小为预先定义的 k 的顶点覆盖,当且仅当 u 是可满足的。

设

$$u = (u_{1,1} \vee u_{1,2} \vee u_{1,3}) \wedge \cdots \wedge (u_{m,1} \vee u_{m,2} \vee u_{m,3})$$

是一个三元合取范式公式,其中每个 $u_{i,j}$ ($1 \leq i \leq m, 1 \leq j \leq 3$) 是布尔变量集合 $V = \{x_1, \dots, x_n\}$ 上的文字。符号 $u_{i,j}$ 用于表示三元合取范式公式中某个文字的位置;第一个下标表示语句,第二个下标表示文字在语句中的位置。归约包括根据三元合取范式公式构造一个图,其中 u 的可满足性与存在一个 G 的包含 $n + 2m$ 个顶点的覆盖是等价的。

为了把满足真值赋值的存在问题转换为一个顶点覆盖问题,我们必须把真值赋值和公式表示为图。我们引入三个边的集合来从三元合取范式公式构造图:设定边的真值集合 T 为真值赋值建模,语句图 C_k 表示语句 u ,链接边 L_k 连接语句图和真值赋值。

G 的顶点包括集合:

- i) $\{x_i, \neg x_i \mid 1 \leq i \leq n\}$, 且
- ii) $\{u_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$ 。

G 的边的集合是设定边的真值、语句边和链接边的并集:

$$T = \{[x_i, \neg x_i] \mid 1 \leq i \leq n\}$$

$$C_k = \{[u_{k,1}, u_{k,2}], [u_{k,2}, u_{k,3}], [u_{k,3}, u_{k,1}]\} \quad (1 \leq k \leq m)$$

$$L_k = \{[u_{k,1}, v_{k,1}], [u_{k,2}, v_{k,2}], [u_{k,3}, v_{k,3}]\} \quad (1 \leq k \leq m),$$

其中 $v_{k,i}$ 是出现在公式中位置 $u_{k,i}$ 的 $x_i, \neg x_i \mid 1 \leq i \leq n$ 的文字。我们首先考虑 T 和 C_k 定义的图的形式以及为了覆盖它们所需要的集合的大小。

T 中的一条边连接一个正文字 x_i 以及其对应的负文字 $\neg x_i$:



一个顶点覆盖必须包含每个 $x_i, \neg x_i$ 对的一个顶点。为了覆盖 T 中的边, 最少需要 n 个顶点。一个有 n 个顶点的 T 的顶点覆盖正好选择 x_i 和 $\neg x_i$ 的某一个。反过来, 这可以被看作是定义了一个 V 上的真值赋值。

每个语句 $u_{j,1} \vee u_{j,2} \vee u_{j,3}$ 生成一个以下形式的子图 C_j 。子图 C_j 连接文字 $u_{j,1}, u_{j,2}$ 和 $u_{j,3}$ 。一个覆盖 C_j 的顶点集合至少包括两个顶点。因此, 集合 T 和 C_k 的边的覆盖至少包含 $n+2m$ 个顶点。

L_j 中的边连接了表示符号 $u_{j,i}$, $u_{j,i}$ 表示文字相对于公式中对应的文字 x_i 或 $\neg x_i$ 的位置。图 16-1 给出了从公式 $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$ 得到的图。很容易看出, 图的构造多项式地依赖于公式中变量和语句的数量。其余部分将说明公式 u 是可满足的, 当且仅当相关的图由一个大小为 $n+2m$ 的覆盖。

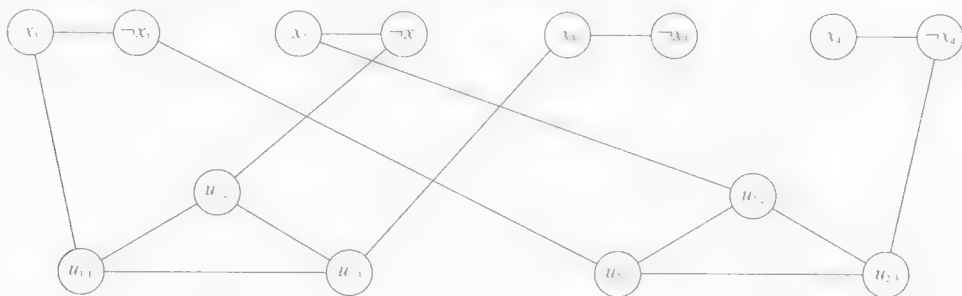
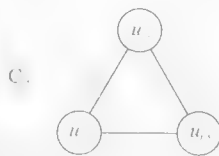


图 16-1 $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$ 的归约表示图

首先, 我们说明一个大小为 $n+2m$ 的覆盖 VC 定义了 V 上的一个满足公式 u 的真值赋值。通过前面的标记我们可以知道, 每一个覆盖至少包括 $n+2m$ 个顶点。因此, 正好是 $x_i, \neg x_i$ 的一个顶点和每个子图 C_j 的两个顶点在 VC 中。我们可以从 VC 得到真值赋值

$$t(x_i) = \begin{cases} 1 & \text{如果 } x_i \in VC \\ 0 & \text{否则} \end{cases}$$

也就是说, t 把顶点覆盖中那些来自 $x_i, \neg x_i$ 对的顶点赋值为 1。

为了说明 t 满足每个语句, 我们考虑子图 C_j 的覆盖。 $u_{j,1}, u_{j,2}$ 和 $u_{j,3}$ 中只有两个顶点可以在 VC 中。假设 $u_{j,k}$ 不在 VC 中, 那么边 $[u_{j,k}, v_{j,k}]$ 必然被 VC 中的 $v_{j,k}$ 覆盖。这说明 $t(u_{j,k}) = 1$ 且该语句被满足。

现在假设 $t: V \rightarrow \{0, 1\}$ 是一个满足 u 的真值赋值。我们可以通过真值赋值构造关联的图的顶点覆盖 VC 。如果 $t(x_i) = 1$, 那么 VC 包含顶点 x_i ; 如果 $t(x_i) = 0$, 那么 VC 包含 $\neg x_i$ 。设 $u_{j,i}$ 是可以被 t 满足的语句 j 的文字。边 $[u_{j,i}, v_{j,i}]$ 被 $v_{j,i}$ 覆盖。添加 C_j 的其他两个边即可完成这个覆盖。显然, $\text{card}(VC) = n+2m$, 如题。

现在我们回过头看看我们的老朋友哈密尔顿回路问题。这个问题已经被证明是确定型机器在指数

时间内可解的 (例 15.5.1), 并且是非确定型机器在多项式时间可解的 (例 15.5.2) 下表中的归约说明哈密尔顿回路问题是 NP-完全的。

归 约	输 入	条 件
三元可满足性	三元合取范式公式 u	u 是可满足的
到	↓	当且仅当
哈密尔顿回路问题	有向图 $G=(N, A)$	G 有周游

由于一个公式的可满足性可以通过检查可能的真值赋值来判定, 因此归约必须将真值赋值表示为图的形式。证明首先定义了子图, 其中的周游对应于真值赋值。

定理 16.3.2 哈密尔顿回路问题是 NP-完全的。

证明: 从三元可满足性到哈密尔顿回路问题的归约可以通过从三元合取范式公式 u 构造有向图 $G(u)$ 完成。我们设计这样的构造, 使得 $G(u)$ 中哈密尔顿回路的存在性等价于 u 的可满足性。设 $u = w_1 \wedge w_2 \wedge \cdots \wedge w_m$ 是一个三元合取范式公式, $V = \{x_1, x_2, \dots, x_n\}$ 是 u 中出现的变量的集合。U 的第 j 个语句表示为 $u_{j,1} \vee u_{j,2} \vee u_{j,3}$, 其中 $u_{j,k}$ 是 V 上的文字。

对于每一个变量 x_i , 设 r_i 是 x_i 在 u 中出现次数和 $\neg x_i$ 在 u 中出现次数中比较大的值。如图 16-2 (a) 所示, 为每个变量 x_i 构造一个图 V_i 。顶点 e_i 可以看作是 V_i 的入口, o_i 是出口。 V_i 中正好有两条开始于 e_i 结束于 o_i 并且正好访问每个顶点一次的路径, 如图 16-2 (b) 和 16-2 (c) 所示。从 e_i 到 $t_{i,0}$ 或 $f_{i,0}$ 的决定了 V_i 中其余的边。

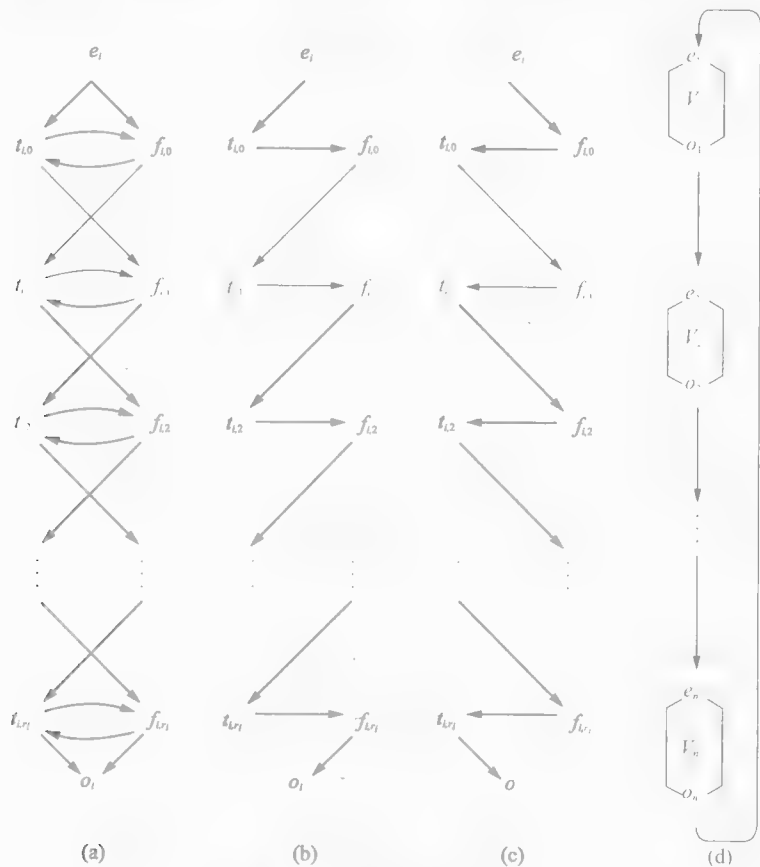


图 16-2 每个变量 x_i 的子图

如图 16-2 (d) 所示, 我们可以连接子图 V_i 来构造一个图 G' 。 V_i 中的两条路径组合生成 2^n 条图 G' 的哈密尔顿回路。图 G' 的每一条哈密尔顿回路表示了 V 上的一个真值赋值。 x_i 的值是由开始于 e_i 的边描述的一条从 e_i 到 $t_{i,0}$ 的边为 x_i 的真值指派为 1。否则, x_i 被赋值为 0。图 16-3 是根据下面的公式构造的:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4)$$

图中粗体边表示的周游定义了一个真值赋值 $t(x_1) = 1$ 、 $t(x_2) = 0$ 、 $t(x_3) = 0$ 和 $t(x_4) = 1$ 。 G' 的哈密尔顿回路对 V 的可能的真值赋值进行编码。现在, 我们证明使用对三元合取范式公式的语句编码的子图来证明 G' 。

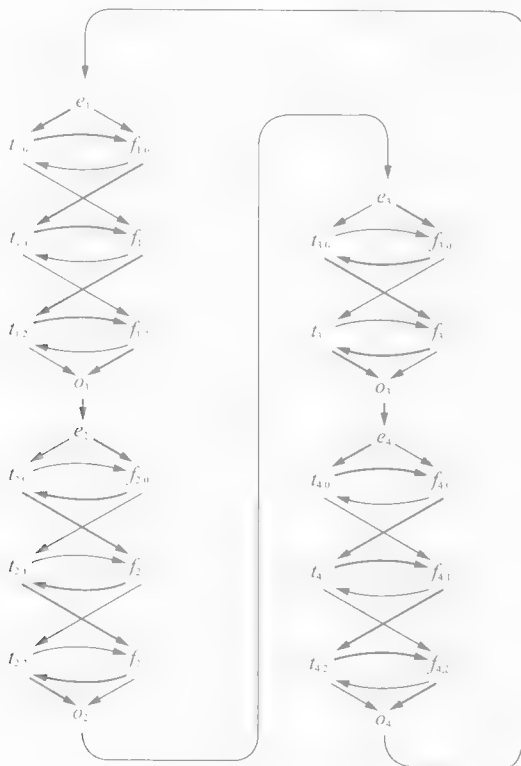


图 16-3 哈密尔顿回路的真值赋值

我们对每个语句 w_i 构造一个图 16-4 形式的子图 C_i 。我们通过按照以下方法连接 G' 的子图来构造图 $G(u)$ 。

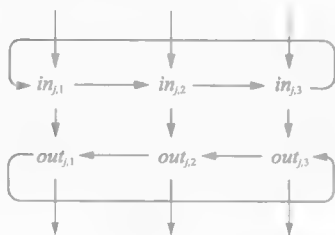


图 16-4 表示语句 w_i 的子图

- i) 如果 x_i 是 w_j 中的文字, 那么选择以前没有被连接到图 C_j 的 $f_{i,k}$ 。在 C_j 中添加一条没有连接到 G' 的、从 $f_{i,k}$ 到顶点 $in_{j,m}$ 的边。然后添加从 $out_{j,m}$ 到 $t_{i,k+1}$ 的边。

- ii) 如果 $\neg x_i$ 是 w_j 中的文字, 那么选择以前没有被连接到图 C_j 的 $t_{i,k}$ 在 C_j 中添加一条没有连接到 G' 的从 $t_{i,k}$ 到顶点 $in_{j,m}$ 的边。然后添加从 $out_{j,m}$ 到 $f_{i,k+1}$ 的边。

图 16-5 是通过把表示语句 $(x_1 \vee x_2 \vee \neg x_3)$ 的子图连接成图 16-3 的 G' 得到的。

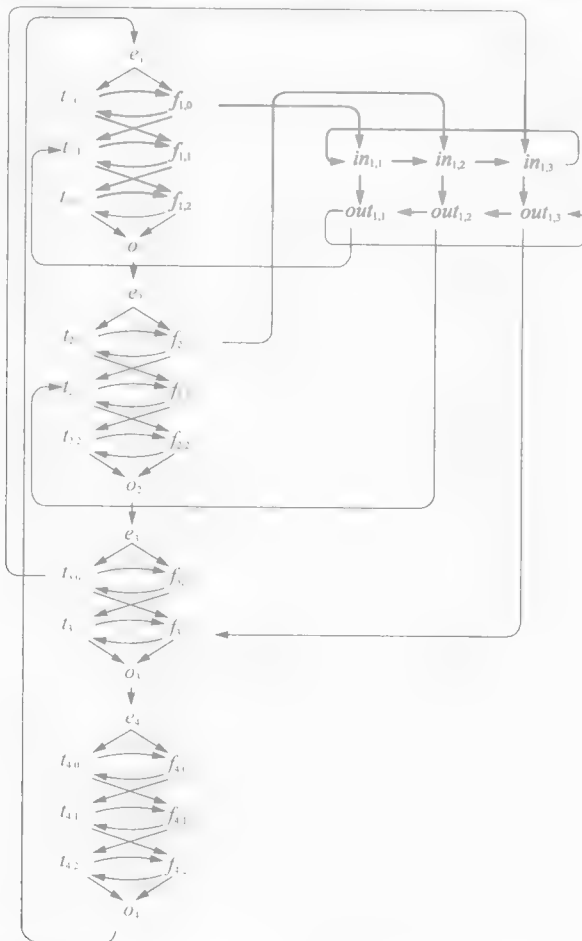


图 16-5 从 C 到 G' 的连接

在图 G' 中一条哈密尔顿回路表示一个真值赋值。如果 x_i 是 w_j 中的正文字, 那么存在一条从某个顶点 $f_{i,k}$ 到 C_j 的某个 in 顶点的边。相似的, 如果 $\neg x_i$ 是 w_j 中的负文字, 那么存在一条从某个顶点 $t_{i,k}$ 到 C_j 的某个 in 顶点的边。当相关的真值赋只能满足 u 时, 我们可以利用这些边把 G' 中的哈密尔顿回路扩展为 $G(u)$ 中的周游。

设 t 是 V 上的一个满足 u 的真值赋值。我们将根据 t 的值构造一个穿过 $G(u)$ 的哈密尔顿回路。我们首先构造一个穿过 v_i 的表示 t 的周游。然后, 再绕道看一下穿过了那些为语句编码的子图的路径。 V_i 路径的一条边 $[t_{i,k}, f_{i,k}]$ 表示真值赋值 $t(x_i) = 1$ 。如果路径通过边 $[t_{i,k}, f_{i,k}]$ 到达节点 $f_{i,k}$, $f_{i,k}$ 没有连接到语句图中, 并且 $f_{i,k}$ 包含一条不在当前路径中的到子图 C_j 的边, 那么按照以下方法将 C_j 连接到周游中:

- i) 通过 C_j 中从 $f_{i,k}$ 到 $in_{j,m}$ 的边绕道至 C_j 。
- ii) 访问 C_j 中的每个顶点一次。
- iii) 通过从 $out_{j,m}$ 到 $t_{i,k+1}$ 的边返回到 V_i 。

绕道到 C_j 说明 G' 中编码的真值赋值满足语句 w_j 。

另一方面, 当 $t(x_i) = 0$ 时, 语句也可以被负文字 $\neg x_i$ 满足。我们可以从顶点 $t_{i,k}$ 构造一个类似的迂回。由于 $t(x_i) = 0$, 所以边 $[f_{i,k}, t_{i,k}]$ 进入顶点 $t_{i,k}$ 。选择一个没有连接到任何一个子图 C_j 的顶点 $t_{i,k}$, 按照以下方式构造迂回:

- i) 通过 C_j 中从 $t_{i,k}$ 到 $in_{j,m}$ 的边绕道至 C_j 。
- ii) 访问 C_j 中的每个顶点一次。
- iii) 通过从 $out_{j,m}$ 到 $f_{i,k+1}$ 的边返回到 V_i 。

由于每个语句都被真值赋值满足, 因此我们可以构造一个访问每个子图 C_j 的 G' 的迂回。通过这种方式, 我们可以把一个满足的真值赋值扩展成 $G(u)$ 中的周游, 从而定义了 G' 的哈密尔顿回路。

现在我们假设图 $G(u)$ 包含哈密尔顿回路。我们必须说明 u 是可以被满足的。哈密尔顿回路按照下述方式定义真值赋值:

$$t(x_i) = \begin{cases} 1 & \text{如果边}[e_i, t_{i,0}] \text{在周游中} \\ 0 & \text{如果边}[e_i, f_{i,0}] \text{在周游中} \end{cases}$$

如果 $t(x_i) = 1$, 那么所有边 $[t_{i,k}, f_{i,k}]$ 都在周游中。另一方面, 如果 $t(x_i) = 0$, 那么周游包含边 $[f_{i,k}, t_{i,k}]$ 。

在证明 t 满足 u 前, 我们首先检查进入子图 C_j 的周游的几个属性。在顶点 $in_{j,m}$ 进入周游时, 这条路径可能访问两个、四个或者 C_j 的所有六个顶点。除了那些在 $out_{j,m}$ 退出的路径, 其他路径都不能成为一个周游的子路径。假设在 $in_{j,1}$ 进入 C_j ; 由于在不访问某些顶点两次的情况下, 下面列出的一些顶点将不能被访问到, 因此下面的 C_j 中的路径不是一个周游的子路径。

路 径	不能到达的顶点
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}$	$out_{j,2}, out_{j,1}$
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}, out_{j,1}, out_{j,3}$	$in_{j,3}$

因此, 惟一从 $in_{j,1}$ 进入 C_j 的周游的子路径必须从 $out_{j,1}$ 退出。同样的性质对顶点 $in_{j,2}$ 和 $in_{j,3}$ 也成立。

周游必须进入每一个 C_j 。如果一条始于顶点 $f_{i,k}$ 的边从顶点 $in_{j,m}$ 进入 C_j , 那么周游通过从 $out_{j,m}$ 到 $t_{i,k}$ 的边退出 C_j 。图 $G(u)$ 中边 $[f_{i,k}, in_{j,m}]$ 的出现说明 C_j 编码的语句 w_j 包含文字 x_i 。此外, 当边 $[f_{i,k}, in_{j,m}]$ 进入 C_j 时, 边 $[t_{i,k}, f_{i,k}]$ 必须进入顶点 $f_{i,k}$ 。另一方面, 顶点 $t_{i,k}$ 不在周游中。因为 $[t_{i,k}, f_{i,k}]$ 在周游中, 所以我们可以得到结论 $t(x_i) = 1$ 。因此, w_j 被 t 满足。类似的, 如果边 $[t_{i,k}, in_{j,m}]$ 进入 C_j , 那么 $\neg x_i$ 在 w_j 中且 $t(x_i) = 0$ 。

综合以上的发现, 我们可以看到通过 $G(u)$ 的哈密尔顿回路生成的真值赋值满足 u 的任意一个语句, 因此满足 u 。接下来只需说明 $G(u)$ 的构造是公式 u 包含的文字数量的多项式。子图 V_i 的顶点和边的数量随着变量 x_i 在语句 u 中出现的次数线性增长。对于每个语句, C_j 的构造为 $G(u)$ 添加 6 个顶点和 15 条边。

许多问题都会将数值和对象关联起来: 开销、重量、价值, 等等。我们在这一部分要考虑的最后一个问题是, 处理数值集合的累计或评估的问题可以是 NP-完全的。该问题的一个古怪的例子是由一个去购物狂欢想花光身上每一分钱的人提出的。问题是: 是否存在一个对象集合使得它们的总花费正好等于该人拥有的钱的总数? 子集一和问题 (subset-sum problem) 对上面的例子进行了形式化。一个子集一和问题的实例包括一个集合 S , 一个值函数 $v: S \rightarrow \mathbb{N}$ 以及一个整数 k 。如果存在一个子集 $S' \subseteq S$ 使得 S' 中所有元素值的和等于 k , 那么该问题的答案是肯定的。为了简单起见, 我们设 $v(A)$ 表示一个集合 A 中所有元素的值的总和。

子集一和问题显然在 NP 内。一个非确定型的猜想可以选择 S 的一个子集。其余的计算累计子集项的值并比较它们的和与问题定义中给出的值 k 。接下来我们只需说明子集一和问题是 NP-难的。

定理 16.3.3 子集一和问题是 NP-完全的。

证明: 从三元可满足性问题到子集一和问题的归约如下表所示。

归 约	输 入	条 件
三元可满足性 到 子集一和问题	三元合取范式公式 u \downarrow 集合 S 、函数 $v: S \rightarrow \mathbb{N}$ 、整数 k	u 是可满足的 当且仅当 存在一个子集 $S' \subseteq S$ [$\sum_{i \in S'} v(i) = k$]

我们需要从三元合取范式公式 u 构造一个集合 S 、一个 S 上的函数 v 以及一个整数 k ，使得 S 有一个元素和为 k 的子集，当且仅当 u 是满足的。跟前面的问题一样，我们设 $u = w_1 \wedge w_2 \wedge \cdots \wedge w_m$ 是一个二元合取范式公式， $V = \{x_1, x_2, \dots, x_n\}$ 是 u 中出现的变量的集合。

集合 S 包含以下各项：

- i) $x_i, i=1, \dots, n$,
- ii) $\neg x_i, i=1, \dots, n$,
- iii) $y_j, j=1, \dots, m$,
- iv) $y'_j, j=1, \dots, m$.

因此 S 有 $2n+2m$ 个对象。我们必须为 S 中的每个对象赋值。每个值都是一个 $n+m$ 位的整数。赋值规则如下：

x_i : 右边第 i 位为 3，如果 x_i 在语句 w_j 中，那么右边第 $n+j$ 位为 1，其他位是 0，

$\neg x_i$: 构造同 x_i ，

y_j : 右边第 $n+j$ 位为 1，其他位是 0，

$\neg y_j$: 同 w_j 。

整数 k 有 $n+m$ 位且每一位都是 3。用这种方式从三元合取范式公式 u 获得的子集一和问题叫做 $S(u)$ 。

为了说明这种构造方法的动机，我们在一个 $(2n+2m) \times (m+n)$ 表中考虑那些入口的对象值的位。

	w_m	...	w_1	x_n	...	x_2	x_1
x_1	—	...	—	0	...	0	3
$\neg x_1$	—	...	—	0	...	0	3
x_2	—	...	—	0	...	3	0
$\neg x_2$	—	...	—	0	...	3	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
x_n	—	...	—	3	...	0	0
$\neg x_n$	—	...	—	3	...	0	0
y_1	0	...	1	0	...	0	0
y'_1	0	...	1	0	...	0	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
y_2	1	...	0	0	...	0	0
y'_2	1	...	0	0	...	0	0

每个值的位置对应于表的 $n+m$ 列。最初 $2n$ 行的入口包含了赋给文字的位。最靠右边的 n 列用于描述真值赋值。最左边的 m 列表示一个文字是否在一个语句中出现。

当 x_i 在语句 w_j 中出现且 $\neg x_i$ 没有出现时，表中与文字 x_i 和 $\neg x_i$ 相关的形式如下：

	w_m	...	w_j	...	w_1	x_n	...	x_i	...	x_1
x_i :	—	...	1	...	0	0	...	3	...	0
$\neg x_i$:	—	...	0	...	0	0	...	3	...	0

列中与语句 w_j 关联的 1 和 x_i 对应的行表示， x_i 在语句中出现，因此，如果 $t(x_i) = 1$ 那么 w_j 被满足。位置 $\neg x_i, w_j$ 处的 0 表示 $\neg x_i$ 没有在 w_j 中出现。

在证明前面的构造是一个从三元可满足性问题到子集和问题问题的归约之前,我们先考虑一下从三元合取范式公式生成的三元可满足性问题实例

$$u = w_1 \wedge w_2 = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4).$$

相应的集合 S 是 $\{x_1, x_2, x_3, x_4, \neg x_1, \neg x_2, \neg x_3, \neg x_4, y_1, y_1', y_2, y_2'\}$, 下面的列表给出了对 S 中每个对象的赋值。

	w_2	w_1	x_4	x_3	x_2	x_1	值
x_1	0	1	0	0	0	3	$v(x_1) = 010003$
$\neg x_1$	1	0	0	0	0	3	$v(\neg x_1) = 100003$
x_2	0	1	0	0	3	0	$v(x_2) = 010030$
$\neg x_2$	0	0	0	0	3	0	$v(\neg x_2) = 000030$
x_3	1	0	0	3	0	0	$v(x_3) = 100300$
$\neg x_3$	0	1	0	3	0	0	$v(\neg x_3) = 010300$
x_4	0	0	3	0	0	0	$v(x_4) = 003000$
$\neg x_4$	1	0	3	0	0	0	$v(\neg x_4) = 103000$
y_1	0	1	0	0	0	0	$v(y_1) = 010000$
y_1'	0	1	0	0	0	0	$v(y_1') = 010000$
y_2	1	0	0	0	0	0	$v(y_2) = 100000$
y_2'	1	0	0	0	0	0	$v(y_2') = 100000$

根据我们对 $S(u)$ 的定义, $k=33333$ 。

公式 u 可以被真值赋值 $t(x_1)=1$ 、 $t(x_2)=1$ 、 $t(x_3)=0$ 和 $t(x_4)=0$ 满足。被真值赋值满足的文字 x_1 、 x_2 、 $\neg x_3$ 和 $\neg x_4$, 以及 y_1 和 y_2 组成的子集能够肯定地回答子集和问题。也就是说, 这些元素的和是 k 。这个例子说明了集合中 y_i 和 y_i' 的角色。当语句 w_i 能被它的一个或者两个文字满足时, 这些对象可以被添加到集合中, 使得与 w_i 关联的列的和为 3。

表中的值说明, 使用语句标号的列中位的和是 5, 使用变量标号的列中位的和是 6。因此, 当给任何 S 的子集添加对象值时, 列之间都不存在传送和交互。

首先, 我们说明如果三元合取范式公式 u 是可满足的, 那么 $S(u)$ 有一个子集且其对象值的和是 k 。设 $t: V \rightarrow \{0, 1\}$ 是一个满足 u 的真值赋值。我们将从这个真值赋值构造一个子集 S' 。最初, 对于每个变量 x_i , S' 包含 x_i 或者 $\neg x_i$; 如果 $t(x_i)=1$, 那么 S' 包含 x_i , 如果 $t(x_i)=0$, 则 S' 包含 $\neg x_i$ 。由于每个 x_i 仅在集合中以正文字或者负文字的形式出现一次, 因此这些对象值的和的最右边 n 位都是 3。

每个语句 w_i 必须被某些文字满足。在与 w_i 关联的列中, 文字的值是 1。因此 w_i 列中对应于真值赋值的文字的位的和最少是 1 最多是 3。如果和是 1, 我们把 y_i 和 y_i' 添加到 S' 中。如果和是 2, 那么我们仅把 y_i' 添加到 S' 中。在以上可能的 y_i 和 y_i' 添加以后, w_i 列中位的和为 3。

设 $S(u)$ 是通过上述构造得到的子集和问题问题的一个实例, 它有一个和为 k 的子集 S' 。我们必须证明 u 是可满足的。首先需要说明的是, x_i 和 $\neg x_i$ 中的一个在 S' 中, 而不是两个都在 S' 。如果它们都不在集合中, 那么 S' 的对象值的和在右边位的第 i 个位置为零。如果 x_i 和 $\neg x_i$ 都在 S' 中, 那么该位置处和为 6。因此, 文字在 S' 中出现的次数定义了真值赋值

$$t(x_i) = \begin{cases} 1 & \text{如果 } x_i \in S' \\ 0 & \text{否则。} \end{cases}$$

对于每个语句 w_i , w_i 列中 S' 的对象值的和是 3。这个总和可以包括 y_i 和 y_i' 的最大值。因此 w_i 列存在一个值为 1 的文字且该文字满足语句 w_i 。■

16.4 归约和子问题

上面介绍的每一个归约都将问题从一个领域转换到另一个不相关的领域: 三元合取范式公式到顶点覆盖, 路径生成和对象集合的值的分析。领域之间的归约要求具有把第一个领域的问题重新配置为等价的第二个领域的问题的能力。这样的转换并不总是明显的或者直接的。幸运的是, 大多数 NP-完

全性证明不要求领域的改变。我们已经给出了同领域问题之间的归约的例子——可满足性到三元可满足性。这些问题的领域是布尔公式的满足和简单的从公式到公式的转换。

说明一个问题是 NP-完全的最普通的技术就是从已知的大量 NP-完全问题中找到一个类似的问题。最重要的规则是，问题越相似，归约需要做的工作就越少。证明问题 P 是 NP-难的最理想的方式是，找到一个 P 的子问题的 NP-完全问题 Q，或者一个能容易地转换成 Q 问题实例的 P 问题实例。我们将通过对前面介绍的 NP-完全问题的归约来说明这些策略。证明过程既不包括设计一个在多项式时间内解决该问题的非确定型算法，也不包括归约可以在多项式时间内完成的证明。根据问题的定义和归约的转换，NP-完全性证明的这两个基本元素是显而易见的。

下面是均分问题 (partition problem)。

均分问题

输入：集合 A，函数 $v: A \rightarrow N$

输出：是；如果存在一个 A 的子集 A' 使得 $v(A') = v(A - A')$

否；否则

询问是否一个集合的元素可以被分为两个不相交的值相等的子集。均分问题和子集一和问题的结果都是由值等于预先给定值的对象集合的存在性决定的。我们可以利用这种相似性来把子集一和问题归约到均分问题，从而说明它是 NP-完全的。

定理 16.4.1 均分问题是 NP-完全的。

证明：从子集一和问题到均分问题的归约

归 约	输 入	条 件
子集一和问题	集合 S，函数 $v: S \rightarrow N$ ，整数 k	存在一个子集 $S' \subseteq S$ 且 $v(S') = k$
到	↓	当且仅当
均分问题	集合 S，函数 $v': A \rightarrow N$	存在一个子集 $A' \subseteq A$ 且 $v'(A') = v'(A - A')$

要求构造一个子集一和问题的实例的集合 A、一个元素 S 和 v 上的值函数 v' 以及 k。集合 A 和值函数 v' 可以如下定义：

$$\begin{aligned} A &= S \cup \{y, z\} \\ v'(x) &= 2v(x), x \in S \\ v'(y) &= 3t - 2k \\ v'(z) &= t + 2k, \end{aligned}$$

其中， $t = v(S)$ 是集合 S 中所有元素的值的和。 $v(x)$ 乘以 2 的原因是要保证集合 A 的值的总和是偶数，只有这样划分才是可能的。使用函数 v' 可知，集合 S 中所有元素的值的总和是 $2t + (3t - 2k) + (t + 2k) = 6t$ 。

首先要说明的是，我们可以根据子集一和问题的解为均分问题构造一个解。由于 S' 是一个解，我们知道

$$v(S') = \sum_{x \in S'} v(x) = k$$

我们定义 A' 是集合 $S' \cup \{y\}$ ，可得：

$$\begin{aligned} v'(A') &= \sum_{a \in A'} v'(a) \\ &= v'(y) + \sum_{x \in S'} v'(x) \\ &= 3t - 2k + 2k \\ &= 3t, \end{aligned}$$

它是 A 的值的总和的一半。因此， A' 是均分问题的解。

现在假设 A 和 v' 可以从 S 、 v 和 k 的归约获得, A 有划分子集 X 和 Y , 且 $v'(X) = v'(Y) = 3t$, 我们必须说明存在一个子集 S' , 其元素的值的总和是 k 。

由于 $v'(y) + v'(z) = 4t$ 大于 A 的值的总和的一半, 因此元素 y 和 z 不可能属于 A 的划分的同一个集合。假设 y 在其中的一个集合 A 中, 那么

$$\begin{aligned} v'(X - \{y\}) &= v'(X) - v'(y) \\ &= 3t - (3t - 2k) \\ &= 2k. \end{aligned}$$

可得 $X - \{y\}$ 是 S 的一个子集, 且它的值 $v(X - \{y\}) = k$ 。因此 $X - \{y\}$ 是子集一和问题的一个解。■

我们来考虑一下学校校规的困境, 它希望成立一个包括学校中每一个俱乐部代表的理事会。事实是一共有 15 个俱乐部并且一个学生可以是任意一个俱乐部的成员。校规要求理事会有 10 个成员。是否可以成立一个满足要求的理事会呢? 这个问题是碰集问题 (hitting set problem) 的一个例子。形式化地说, 一个碰集问题的实例包括一个集合 S , 一个 S 的子集的有限族 (collection) $\mathcal{C} = \{C_1, \dots, C_n\}$, 以及一个整数 k 。如果对于每个 C_i 都有 $C \cap C_i \neq \emptyset$, 那么集合 C 是 \mathcal{C} 的碰集。也就是说, 每个集合 C_i 都被 C 的元素击中。如果存在一个大小等于或小于 k 的碰集, 那么该问题的答案是肯定的。

我们也可以不考虑 C 的元素击中 C_i , 而是考虑元素覆盖 C_i 。这种解释揭示了顶点覆盖问题和碰集问题的相似性。我们会把顶点覆盖问题归约到碰集问题, 从而说明碰集问题是 NP-完全的。

定理 16.4.2 碰集问题是 NP-完全的。

证明: 我们可以按照下述方式, 从顶点覆盖问题的一个实例 $G = (N, A)$, k 得到碰集问题的一个实例 S 。元素是 G 的节点。每条边 $[n_i, n_j]$ 定义了一个包含两个元素的集合 $\{n_i, n_j\}$ 。类 \mathcal{C} 包括所有从 G 的边得到集合。最后, k 在这两个问题中是相同的。现在我们来证明, G 有一个大小为 k 的顶点覆盖, 当且仅当, 存在一个规模小于等于 k 且与类 \mathcal{C} 相关的碰集。

假设存在一个大小为 k 的顶点覆盖 VC 。该集合是一个适当大小的 \mathcal{C} 的碰集。反之, 假设 \mathcal{C} 有一个小于等于 k 的碰集 C 。则每一个集合 $\{n_i, n_j\} \in \mathcal{C}$ 都被 C 的元素击中。在图 G 的表示中, 就是每条边 $[n_i, n_j]$ 都被 C 中的一个顶点覆盖。因此 C 是不大于 k 的顶点覆盖。■

通过把边解释成包含两个元素的集合和把覆盖解释成击中, 顶点覆盖问题就变成了碰集问题的子问题。把一个已知的 NP-完全问题解释成一个待定问题的子问题的能力通常可以使得 NP-完全性的证明显得几乎微不足道。装箱问题 (bin-packing problem) 是这种情况的另外一个例子。我们将说明均分问题可以简单地转换成装箱问题的子问题。

装箱问题

输入: 集合 A , 规模函数 $s: A \rightarrow \mathbb{N}$, 正整数 k 和 m

输出: 是; 如果存在一个 A 的划分 A_1, A_2, \dots, A_k 使得 $s(A_i) \leq m (1 \leq i \leq k)$

否; 否则

定理 16.4.3 装箱问题是 NP-完全的。

证明: 归约形式如下:

归约	输入	条件
均分问题	集合 A , 函数 $v: A \rightarrow \mathbb{N}$	存在一个子集 $A' \subseteq A$ 且 $v'(A') = v'(A - A')$
到	↓	当且仅当
装箱问题	集合 A , 函数 $s = v$, 整数 k 和 m	存在一个划分 A_1, A_2, \dots, A_k , 使得对于所有 i 都有 $s(A_i) \leq m$

正如归约的描述所示, 两个问题使用同样的集合和函数。剩下的问题是要为装箱问题选择整数 k 和 m 。由于均分问题试图把 A 分成两个值相等的集合, 我们设 $k = 2$ 。最后设置 m 等于 A 中元素值的和的一半, 即 $m = s(A)/2$, 归约完成。

[516]

这个归约将均分问题看作是有两个容量最大为 $s(A)/2$ 箱子的装箱问题。如果一个集合 $A' \subseteq A$ 满足均分问题, 那么 A' 和 $A - A'$ 构成了一个满足装箱问题的划分。反之, 装箱问题的一个容量边界为 $s(A)/2$ 的解 A_1 和 A_2 也是均分问题的解。

16.5 最优化问题

有许多问题的目标不仅仅是判定是否存在一个解, 而是找到一个最理想的解。一个最理想的解可以最小化代价, 最大化值, 最高效地利用资源等等。由于其结果不再是一个是或否的回答, 因此最优化问题不符合我们对判定问题的定义。然而, 我们为判定问题考虑的时间复杂性问题在最优化问题中同样存在。

我们使用货郎担问题 (traveling salesman problem) 来说明证明最优化问题 NP-完全性所使用的策略。货郎担问题是哈密尔顿回路问题的一般化, 它试图在带权有向图中找到一个代价最小的周游, 其中一个路径的代价就是路径上边的权的总和。这个问题描述了这样一种情况, 一个货郎希望他路途上的每一村庄正好一次, 并且走最短的路程。

通过为问题实例添加一个距离边界, 我们可以把货郎担问题转换为判定问题:

货郎担问题

输入: 带权有向图 $G = (N, A, w)$, 整数 k

输出: 是; 如果 G 有一个代价小于等于 k 的周游

否; 否则

给代价增加一个边界 k 把期望路径的答案变成了是或否响应。

我们可以迭代地使用判定问题的解来生成一个原问题的解。设 n 是 G 的节点数量, l 是代价最小的 n 条边的代价总和, u 是代价最大的 n 条边的代价总和。任何一个 G 的周游的代价都在 l 和 u 之间。我们可以迭代地对下面一系列判定问题求解直到得到一个肯定答案或者问题实例返回否定响应, 这样便能获得代价最小的周游的代价。在后一种情况下, 图中不存在周游。

$G = (N, A, w), k = l$

$G = (N, A, w), k = l + 1$

$G = (N, A, w), k = l + 2$

\vdots

$G = (N, A, w), k = u$

[517]

定理 16.5.1 货郎担问题是 NP-完全的。

证明: 哈密尔顿回路问题可以看作是货郎担问题的子问题。设 $G = (N, A)$ 是哈密尔顿回路问题的实例。为了获得货郎担问题的实例, 我们仅仅需要为 G 定义一个权重函数 w 和边界 k 。设 w 给每一条边赋值 1, 设 k 是 G 的节点数量。图 G 有一个周游, 当且仅当, 相应的带权有向图 (N, A, w) 有一个代价为 k 的周游。

背包问题 (knapsack problem) 是一个经典的最优化问题, 它关心的是在一定的大小限制下, 选择一个值最大的对象集合。对这个问题的最有趣的描述讲述了这样种情况, 一个小偷必须决定把哪些东西放进他的背包里。他的目标是对象的值最大, 但是他的选择受制于背包的大小。背包问题的判定问题版本是:

背包问题

输入: 集合 S , 规模函数 $s: S \rightarrow \mathbb{N}$, 值函数 $v: S \rightarrow \mathbb{N}$, 最小值 m , 大小边界 b

输出: 是; 如果存在一个子集 $S' \subseteq S$ 使得 $s(S') \leq b$ 且 $v(S') \geq m$,

否; 否则

定理 16.5.2 背包问题是 NP-完全的。

证明: 归约形式如下:

归 约	输 入	条 件
均分问题	集合 A , 函数 $v: A \rightarrow \mathbb{N}$	存在一个子集 $A' \subseteq A$ 且 $v(A') = v'(A - A')$
到	\downarrow	当且仅当
背包问题	集合 A , 函数 $s = v, v$, 整数 b 和 m	存在一个子集 A' 使得 $s(A') \leq b$ 且 $v(A') \geq m$

这个归约在跟均分问题相同的领域内创建了一个背包问题。背包问题的规模函数和值函数都用于设置均分问题的规模函数。把 b 和 m 定义为 $s(A)/2$ 归约即可完成。由于背包问题的规模函数和值函数与均分问题的规模函数相同, 因此一个集合 A' 满足均分问题的要求, 当且仅当它也满足相应的背包问题的要求。

1518

16.6 近似算法

NP 类的重要性不仅仅是理论上的, 而是实际的。NP-完全问题非常自然地出现在很多领域, 包括模式识别、调度、决策分析、组合数学、网络设计和图论等。判定一个问题是 NP-完全的并不意味着不再需要解, 而是非常不可能找到能解决这些问题的多项式时间算法。

我们通过货郎的考虑来考察处理 NP-完全问题的过程。货郎希望他决定路线的过程能够自动化。我们把城市 and 道路表示成有向带权图 $G = (N, A, w)$ 的节点和边, 权重函数 $w(x, y)$ 给出了从城市 x 到城市 y 的道路的距离。货郎必须访问的城市是可以变化的, 此时他必须计算一个新路线。当然, 他的目标是访问每个城市一次然后回家, 并且在旅途中花费尽可能少的时间。已知货郎担问题是 NP-完全的, 货郎应该怎样解决设计路线的问题?

第一步是要判定问题的 NP-完全性是否与他特定的情况相关。如果路线仅包含几个城市, 那么解决这个问题的算法的渐进性能是无关紧要的。构成一个小问题实例的节点数量取决于可用的计算资源和算法应用的频率。

如果算法经常被使用, 即使是对一个相对小的节点数量, 那么它也值得我们使用算法理论的技术来优化查找技术。例 15.5.1 的图灵机用来解决哈密尔顿回路问题的穷尽测试所有节点序列的策略需要检查 n^{n-1} 条可能路径, 其中 n 是城市的数量。分支定界法 (branch-and-bound algorithm) 可以用于裁剪搜索数并减少需要考虑的路径的数量。一个动态程序算法可以在 $O(n2^n)$ 时间生成最小距离周游。尽管仍然是指数级的, 与穷尽搜索策略相比这已经是一个相当可观的缩减了。

如果需要的话, 货郎下一步需要考虑把问题重新公式化为另一个可以在多项式时间解决的问题。新问题的解可能不是最理想的周游, 但是对于他的目的而言这些解是可以接受的。按照这种方法, 货郎在地图上标记他必须访问的所有城市并决定设一个路线, 这个路线从最东边的城市开始然后单独从东到西地走到最西边的城市。当他按照严格的从西到东路线返回到原城市时, 周游结束。

从东到西策略的动机是两个城市之间的最短路线不应该包括偏离目标的分支。这个方法经常产生不错的近似解, 然后, 图 16-6 给出了一个最理想的距离为 82 的周游, 而双向的解的距离是 140。我们可以增加更多从 a_1 到 a_7 的“Z 字路线”来继续图中从节点 a_2 到 a_6 的构成模式。这不会增加最小代价的周游, 但是最小代价的双方向的周游可以被增加到任意大。

1519

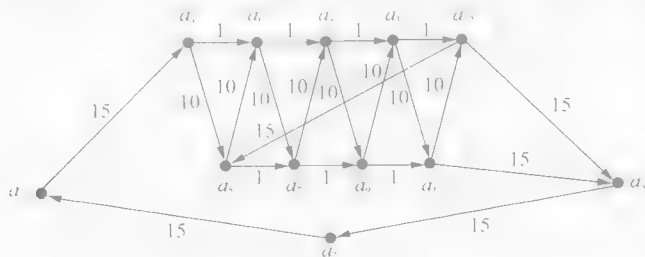


图 16-6 双向货郎担问题的解

认识到他的双方向策略可能产生过长的周游后, 货郎问了以下两个问题:

1. 是否存在一个解决双方向问题的, 又不会导致周游比最理想周游任意大的多项式时间算法?
2. 如果上面问题的答案是否定的, 那么为了获得一个多项式时间的近似解还需要添加哪些条件?

我们将会介绍了近似算法性能度量后再回答这些问题。

最优化问题的解包括一个我们一般用作解的代价的参考的数值。例如, 货郎担问题实例的解包括一个周游以及作为其代价的总距离。背包问题的解包括一个对象集合, 其相关代价是集合内对象的值的和。近似算法可以生成代价可能不是最理想的解。近似算法的错误是最理想解的代价和近似解的代价之前的差别。

设 $c(p_i)$ 和 $c^*(p_i)$ 分别表示最优化问题 P 的实例 p_i 的近似算法产生的解的代价和最优解的代价。近似算法的质量是通过近似解的代价和最优解的代价的对比来度量的。

定义 16.6.1 一个对最优化问题 P 产生近似解的算法叫做 α -近似算法 (α -approximation algorithm), 如果

i) 问题是最小化问题且 $c(p_i) \leq \alpha \cdot c^*(p_i)$, 或者

ii) 问题是最大化问题且 $c^*(p_i) \leq \alpha \cdot c(p_i)$

[520] 对于所有的 $\alpha \geq 1$ 和所有 P 的实例 p_i 都成立。

一个最小化问题的 2-近似算法产生的解的代价至多是最优解的代价的两倍。对于一个最大化问题, 2-近似算法产生的解的代价至少是最优解的代价的一半。

我们可以这样重新阐述货郎问题: “货郎担问题是否存在一个多项式时间 α -近似算法” 和 “为了得到一个多项式时间的 α -近似算法, 问题要做哪些必要的变化?” 第一个问题的答案是否定的, 除非 $P = NP$ 。第二个问题的一个答案是, 如果图是全连通的并且距离满足三角不等式, 那么我们可以得到一个 2-近似算法。

定理 16.6.2 如果 $P \neq NP$, 那么货郎担问题不存在多项式时间 α -近似算法。

证明: 我们将证明货郎担问题的多项式时间 α -近似算法可以用于在多项式时间内解决哈密尔顿回路问题。因为如果 $P \neq NP$, 那么后者是不可能做到的。因此可知在同样的假设下不存在这样的算法。

我们首先定义哈密尔顿回路问题实例到货郎担问题实例的转换。设 $G = (N, A)$ 是哈密尔顿回路问题的实例且 $n = \text{card}(N)$ 。相应的货郎担问题是全连通图 $G' = (N, A', w)$, 其中 w 的定义如下:

$$w(x, y) = \begin{cases} 1 & \text{如果 } [x, y] \in A \\ \alpha \cdot n + 1 & \text{否则} \end{cases}$$

显然, 根据 G 构造 G' 的过程可以在 G 的表示长度的多项式时间内完成。

如果 G 是一个周游, 那么相应的 G' 中的周游代价为 n 。如果 G 不包含周游, 那么 G' 中的每一个周游的代价都大于 $\alpha \cdot n$, 这是因为它至少包含一条不在 A 中的边。在前一种情况下, 由于所有其他周游都超出了近似值边界, 因此, 在 G' 上运行一个 α -近似算法必然产生一个代价为 n 的周游。因此, G 包含一个周游, 当且仅当 α -近似算法返回一个代价为 n 的周游。

上述的相等描述了一个哈密尔顿回路问题的解: 从 G 构造 G' 并且使用近似算法获得了一个 G' 的周游。根据以上观察, 近似算法返回的周游的长度为 n , 当且仅当 G 包含周游。如果 α -近似算法在多项式时间内是可计算的, 那么相应的哈密尔顿回路问题的解在多项式时间内也是可计算的。

当图 $G = (N, A, w)$ 是全连通的, 并且其距离函数是可交换的且满足三角不等式, 那么我们可以很容易地为货郎担问题生成一个 2-近似算法。也就是说,

$$\begin{aligned} w(x, y) &= w(y, x) \quad \text{且,} \\ w(x, y) &\leq w(x, z) + w(z, y) \end{aligned}$$

对于所有 $x, y, z \in N$ 都成立。有时候, 带有附加条件的货郎担问题被叫做欧几里得货郎担问题 (euclidean traveling salesman problem)。

近似算法首先构造一个 G 的最小代价生成树。一个无向连通图的生成树是一个包含了图中所有节点的连通的非循环的子图。生成树的代价是树中所有边的权的和。如果一个带权有向图 G 是全连通的

且其距离函数是可交换的, 那么这个图可以被看作是无向图。对于每条边 $[x, y]$, 存在一条同样权重的边 $[y, x]$ 。

把图 G 解释成无向图后, 我们可以用 Prim 算法在时间 $O(n^2)$ 内生成一个最小生成树, 其中 n 是 G 的节点数量。下面这个包含四个步骤的过程定义了一个欧几里得货郎担问题的 2-近似算法。

1. 选择一个节点 $x \in N$ 作为生成树的根节点。
2. 建立 G 的最小代价生成树。
3. 构造生成树的前序遍历所访问的节点序列。
4. 删除在序列中出现多于一次的节点。

图 16-7 给出了从生成树获取周游的过程。一个前序遍历从根节点 c 开始, 访问所有节点 (许多节点多次被访问), 在 c 处结束。为了从遍历生成的路径中获取周游, 我们顺序地删除对同一个节点的多次访问。在图 16-7 的序列中, 节点 c 在 a 后和 d 前被重复访问, 并且三角不等式保证了备用路径不比原有路径长。

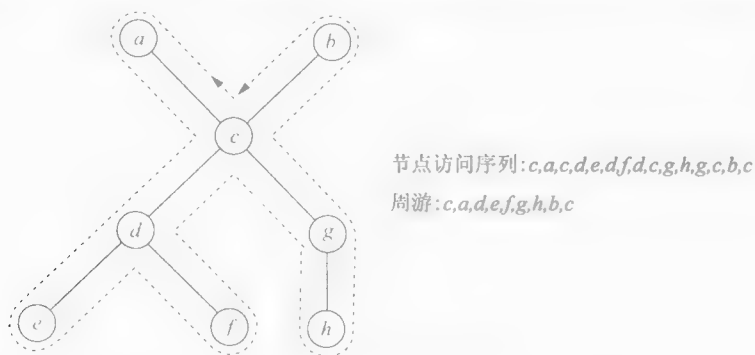


图 16-7 生成树获取周游

除了根节点在路径开始和结束处的出现, 我们可以重复这个过程来剔除所有节点的多次出现。结果路径是一个周游。为了分析周游的代价, 我们设 t^* 、 m^* 、 p 和 t 分别代表最小代价周游、最小代价生成树、前序遍历生成的路径和使用节点剔除策略得到的周游。

我们可以通过删除图的最小代价周游的任何独立的边来得到一个生成树。结果生成树的代价是最小代价生成树 M 的代价的边界。因此,

$$m^* \leq t^*.$$

生成树的每一条边在前序遍历生成的路径都出现两次, 所以

$$p = 2m^*.$$

[522]

由于节点删除过程不会增加结果路径的代价, 因此算法产生的周游的代价以前序路径的代价为边界。综合不等式

$$t \leq p \leq 2t^*,$$

生成了通过这种方式构造的周游的 2-近似边界。

本节我们列出了当遇到 NP-完全问题时构造解的策略。这位虚构的货郎使用的步骤是:

- a) 判定渐近复杂性是否与问题相关,
- b) 把问题重新形式化为一个高效地可解问题, 或者
- c) 开发一种能生成近似解的算法。

这些步骤为我们获取 NP-完全最优化问题的合适的解提供了很好的开端。

16.7 近似方案

一个理想的逼近 NP-完全问题的系统应该允许用户描述在特定应用中所允许的错误的程度。对那些高精度尤其重要的问题, 我们可以选择一个错误边界来帮助达到必要的精度。对于那些不要求高精

523

精确的问题,我们可以使用更加有效的方式来生成不是那么精确的近似解。对于一些 NP-完全问题,这个理想是可以实现的。

一个近似方案 (approximation scheme) 是指这样一个算法,其输入参数用于描述可接受的错误边界。最小化问题的一个参数为 k 的近似方案生成的解对于所有问题实例 p_i 满足

$$c^*(p_i) \leq c(p_i) \leq \frac{k+1}{k} \cdot c^*(p_i)$$

最大化问题的边界是

$$\frac{k}{k+1} \cdot c^*(p_i) \leq c(p_i) \leq c^*(p_i).$$

在其他情况下, k 的值得增加会提高近似的精确度。一个多项式时间近似方案是这样: 一个近似方案, 对于所有变量而言, 其时间复杂性是参数 k 的多项式。

我们将使用背包问题来说明近似方案的属性。背包问题的最简单的近似方案是, 最初在背包中放置一些项, 然后使用贪心算法完成选择。背包问题最优化版本的一个实例包括一个集合 $S = \{a_1, \dots, a_n\}$, 规模函数 $s: S \rightarrow \mathbb{N}$, 值函数 $v: S \rightarrow \mathbb{N}$ 和大小边界 b 。我们分别使用 c^* 和 c 表示背包问题的最优解和近似解的值。

背包问题的贪心算法会选择: 一个适合于背包的相对值 $v(a_i)/s(a_i)$ 最高的项 a_i 。这个过程会一直重复到剩余的项都不能放进背包中。不幸的是, 使用这种方法不能产生错误边界 (练习 14)。

参数为 k 的近似方案按照下述方式选择一个集合:

1. 生成所有势小于等于 k 的子集 $I_i \subseteq S$ 。
2. 对于每个大小满足 $s(I_i) \leq b$ 的子集, 根据初始值、规模函数和边界 $b - s(I_i)$, 在集合 $S - I_i$ 上使用贪心算法生成集合 G_i 。合并集合 I_i 和 G_i 以便生成集合 $T_i = I_i \cup G_i$ 。
3. 集合 T_i 有最大值。

包含元素个数小于等于 k 的子集的生成以及判定它们是否满足边界条件的测试过程能够产生一系列初始集合 I 。可以通过使用贪心算法生成集合 G_i 来完成初始集合。初始集合的总集合 T_i 是 $I \cup G_i$ 。我们需要说明的是, 对于每个问题实例和所有 $k \geq 1$, 算法都能产生一个满足下面条件的近似解:

$$\frac{k}{k+1} \cdot c^* \leq c.$$

524 假设集合 T 给出的最优解有 j 个元素。我们考虑两种情况 $j \leq k$ 和 $j > k$ 。

情况 1: $j \leq k$ 。集合 T 是第一步生成的某个初始集合, 最优解通过算法产生。

情况 2: $j > k$ 。最优解 T 可以划分成两个集合 $I = \{\hat{a}_1, \dots, \hat{a}_k\}$ 和 $R = \{\hat{a}_{k+1}, \dots, \hat{a}_j\}$, 其中 I 包含 T 的 k 个值最大的项, R 包含剩余的项, 这些项按照其相对值排序:

$$v(\hat{a}_{k+1})/s(\hat{a}_{k+1}) \geq v(\hat{a}_{k+2})/s(\hat{a}_{k+2}) \geq \dots \geq v(\hat{a}_j)/s(\hat{a}_j).$$

首先我们需要注意的是, 对于每个 $a \in R$ 都有 $v(a_i) \leq c^*/(k+1)$ 。对于每个值大于 a_i 的 $\hat{a}_i \in I$, 都有 $v(I) \geq k \cdot v(\hat{a}_i)$ 。因此

$$c^* = v(T) = v(I) + v(R) \geq k \cdot v(\hat{a}_i) + v(\hat{a}_i) \geq (k+1)v(\hat{a}_i)$$

成立。

考虑使用贪心算法从集合 I 生成的近似解。如果贪心算法选择了所有在最优解 R 中的项, 那么算法生成一个最优解。

否则, 设 G 是贪心算法生成的 I 的扩展, a_m 是 R 中没有被贪心算法选中的第一个项。只有在考虑 a_m 时背包只剩下一个不充分的空间, 这种情况才会发生。集合包括了 R 中的 $\hat{a}_{k+1}, \hat{a}_{k+2}, \dots, \hat{a}_{m-1}$, 其他项的相对值大于 a_m 。现在, 我们用 G_m 来生成集合 R 的值的上界。 G_m 中的元素的比值大于 R 中那些大小总和为 $s(G_m)$ 的初始项。这是因为 G_m 包含了 R 中所有比值大于 $v(\hat{a}_m)/s(a_m)$ 的对象。 G_m 中所有其他对象的相对值都小于 $v(a_m)/s(a_m)$ 。注意, R 中项的大小不需要正好累计到 $s(G_m)$ 。我们可以考虑分割一个项来获得大小为 $s(G_m)$ 的 R 的子集。

可以添加到 G_m 的可能的最大值最多填满背包 $v(a_m)$ 的空间, 因为剩下的空间小于 $s(a_m)$, 并且贪

心算法在按照相对值顺序搜索时已经通过了 \hat{a}_m 。由于 R 中的项 $\hat{a}_m, \dots, \hat{a}_j$ 的相对值最大是 $v(\hat{a}_m)/s(\hat{a}_m)$ ，所以这也是填补背包剩余空间的值的上界。通过上面的观察，我们可以看到

$$c^* = v(I) + v(R) \leq v(I) + v(G_m) + v(\hat{a}_m) \leq v(I) + v(G) + v(\hat{a}_m).$$

使用不等式 $v(\hat{a}_m) \leq c^*/(k+1)$ ，我们可以得到

$$c^* \leq v(I) + v(G) + v(\hat{a}_m) \leq c + c^*/(k+1)$$

或者

$$\frac{k}{k+1} \cdot c^* \leq c$$

525

我们还需要说明针对每一个 $k \geq 1$ 生成的近似算法是背包问题实例的大小的多项式。设 $C(n, i)$ 是一次拿占用 i 的 n 个东西的并的数量，势小于等于 k 且包含 n 个对象的子集的数量是

$$\begin{aligned} \sum_{i=0}^k C(n, i) &= 1 + \sum_{i=1}^k \frac{n(n-1)\cdots(n-i+1)}{i!} \\ &\leq 1 + \sum_{i=1}^k n^i \\ &\leq 1 + \sum_{i=1}^k n^k \\ &= 1 + k \cdot n^k. \end{aligned}$$

使用贪心算法扩展所有这些需要的时间为 $O(n)$ 。因此时间复杂性是 $O(k \cdot n^{k+1})$ 。

尽管上述近似算法是每个 k 的多项式，但是时间复杂性却随着参数 k 做指数级的增长。因此，错误的降低伴随着生成近似解所需时间的指数级增长。一个是 n 和 k 的多项式的近似方案叫做完全多项式。结合贪心算法和动态程序可以降低时间复杂性，从而可以为背包问题生成一个的 $O(k \cdot n^2)$ 完全多项式近似方案。

16.8 练习

1. 如果一个公式是合取公式，且它的每个语句包含两个文字的析取，那么它是二元合取范式。证明二元合取范式的可满足性问题是 \mathcal{P} 内。
2. 如果一个公式是合取公式，且它的每个语句包含四个文字的析取，那么它是四元合取范式。证明二元合取范式公式的可满足性问题是 NP-完全的。
3. 请为子集和问题设计一个字符串表示来描述能在多项式时间内解决该问题的非确定型图灵机的计算。
4. 请设计一个从均分问题到子集和问题的多项式时间归约。定理 16.4.1 给出了一个从子集和问题到均分问题的多项式时间归约。
5. 无向图 G 的团是一个 G 的子图，该子图中每两个节点都由一条边连接。团问题要解决的是，对于任意一个图 G 和整数 k ， G 是否有一个大小为 k 的团。请证明团问题是 NP-完全的。
提示：为了证明团问题是 NP-难的，需要说明图 G 的团和补图 \bar{G} 的顶点覆盖之间的关系。 \bar{G} 中顶点 x 和 y 之间存在一条边，当且仅当 G 中不存在连接着两个顶点的边。
6. 设 $\mathcal{C} = \{C_1, \dots, C_n\}$ 是 S 的子集的集族。一个子集族 $\mathcal{C}' \subseteq \mathcal{C}$ 覆盖 S 如果

$$S = \bigcup_{C_i \in \mathcal{C}'} C_i.$$

526

最小覆盖问题关心一个集族 \mathcal{C} 是否是这样一个子集族，该子集族能够覆盖 S 且其大小小于等于 k 。请证明最小覆盖问题是 NP-完全的。

7. 设 \mathcal{C} 是有限个集合的集族， k 是一个小于等于 \mathcal{C} 的势的整数。请证明判定 \mathcal{C} 是否包含 k 个不相交的集合的问题是 NP-完全的。
8. 最长路径问题的实例是一个图 $G = (N, A)$ 和整数 $k \leq |A|$ 。请说明判定 G 是否有一个非循环的包括 k 或更多边的路径的问题是 NP-完全的。

- *9. 多处理器调度问题的输入包括一个任务集合 A ，一个描述每个任务执行时间的长度函数 $l: A \rightarrow \mathbb{N}$ ，和可用的处理器个数 k 。该问题的目标是找到一个 A 的划分 A_1, A_2, \dots, A_k ，使其能够最小化完成所有任务所需的时间，也就是说在所有划分上最小化 $\max\{l(A_i) \mid i=1, \dots, k\}$ 。
- 请将多处理器调度问题形式化为一个判定问题。
 - 请说明相关的判定问题是 NP-完全的。
- *10. 整数线性规划问题是：给定一个 $n \times m$ 的矩阵 A 和一个长度为 n 的列向量 b ，是否存在一个列向量 x 使得 $Ax \geq b$ ？请使用一个三元可满足性的归约来证明整数线性规划问题是 NP-难的（整数线性规划问题也在 NP 中；证明要求一些线性代数的基本属性知识）。
11. 请说明无向图的货郎担判定问题是 NP-完全的。
12. 顶点覆盖问题最优化版本的目的是找到一个无向图 G 的最小顶点覆盖。近似策略这样构造一个覆盖 VC：从 G 中选择任意一个边 $[x, y]$ ，添加到 VC 中，从 G 中删除 x, y 和所有与 $[x, y]$ 关联的边，并重复这个选择和删除过程，直到 VC 覆盖了原图。证明顶点覆盖问题的这个策略需要一个多项式时间的 2-近似算法。
- [527] *13. 装箱问题最优化版本的输入包括一个集合 A ，一个规模函数 $s: A \rightarrow \mathbb{N}$ ，以及一个大小为 n 的箱子，其大小大于最大对象的大小。该问题的目的是判定为了存放 A 中的对象所需要的箱子的最小数量，其中箱子大小 n 是可以放在一个箱子内的对象的大小的总和的上界。首次适应算法将对象放在能容纳它的第一个箱子内。如果它不能被当前的箱子容纳，那么把它放在一个新的箱子中。这个过程一直重复直到所有的对象都已经存放。说明装箱问题的首次适应策略能生成一个多项式时间的 2-近似算法。
14. 背包问题的贪心策略用于选择适合背包的相对比值 $v(a)/s(a)$ 最高的项 a 。这个过程一直重复直到没有其他的项可以放进背包中。说明使用贪心选策略时，存在一个可能错误的上界。
- *15. 背包问题的近似解可以通过按照如下方式修改贪心策略获得：算法返回一个贪心算法生成的解，或者一个包含了适应于背包的值最大的项的解。证明这种改动能够为背包问题生成一个 2-近似算法。

参考文献注释

Karp [1972] 的学术论文证明了三元可满足性问题、顶点覆盖问题和哈密尔顿回路问题的 NP-完全性。Garey 和 Johnson [1979] 关于 NP-完全性的经典著作讨论了本章涉及的所有问题以及很多其他问题。这本书还包括了大多数练习需要的归约的描述。

由于 NP-完全问题的重要性，现在已经出现了关于近似算法话题的扩展文化。上述 Garey 和 Johnson 的著作、Papadimitriou、Steiglitz [1982] 和 Hochbaum [1997] 的著作都给出了近似算法领域的介绍。Christofides [1976] 为经典的货郎担问题设计了一个多项式时间的 1.5-近似算法。第 16.6 节介绍的背包问题的近似方案来自 Sahni [1975]。Ibbara 和 Kim [1975] 使用动态程序为背包问题提出了 $O(k \cdot n^2)$ 的完全多项式的近似方案。

还有许多关于一般算法理论的优秀书籍，包括 Cormen、Leiserson、Rivest 和 Stein [2001]、Lebitin [2003]、Brassard 和 Bratley [1996]。除了 NP-完全问题和近似算法，这些著作包括了本章中所介绍的近似算法策略，如图算法、贪心算法和动态程序策略等。

第 17 章 其他复杂性类

复杂性理论关心的是，为了解决判定问题或计算函数，对判定语言成员资格问题所需要的资源进行估计。时间复杂性的研究已经把可以被多项式时间算法求解的类 \mathcal{P} 看作是高效可解问题。我们在本章的开始首先讨论几个可以从类 \mathcal{P} 和类 \mathcal{NP} 推导出来的时间复杂性类的属性。然后研究计算所需要的时间和空间之间的关系。最后，我们使用空间复杂性来说明存在不能被任何多项式时间算法或多项式空间算法解决的问题。

17.1 派生的复杂性类

关于易解性的研究介绍了类 \mathcal{P} 的语言在多项式时间内是确定地可判定的，通过非确定的计算，类 \mathcal{NP} 的语言也是多项式时间可判定的。这两种情况是否相同的问题目前仍是未知的。现在我们考虑其他几个能够帮助更好地考察 $\mathcal{P} = \mathcal{NP}$ 问题的语言类。非常有趣的是，这些类的属性通常依赖于 \mathcal{P} 和 \mathcal{NP} 之间的关系。下面的讨论主要是在 $\mathcal{P} \neq \mathcal{NP}$ 假设下进行的。然而，在任何使用了这个假设的结果中，这个条件将会被显示地叙述。

类 \mathcal{P} 和 \mathcal{NPC} 都是 \mathcal{NP} 的非空子集，但是这两个类之间有什么关系？根据定理 15.6.2，如果 $(\mathcal{P} \cap \mathcal{NPC})$ 非空，那么 $\mathcal{P} = \mathcal{NP}$ 。因此，在 $\mathcal{P} \neq \mathcal{NP}$ 的假设下， \mathcal{P} 和 \mathcal{NPC} 一定不相交。15.9 节的图说明了 $\mathcal{P} \neq \mathcal{NP}$ 时 \mathcal{P} 和 \mathcal{NPC} 的内容。当我们看到这个图时，立即会想到这样一个问题： \mathcal{NP} 中是否存在不在 \mathcal{P} 或者 \mathcal{NPC} 中的语言？ [529]

我们定义一个语言族 \mathcal{NPJ} 来表示所有在 \mathcal{NP} 但是不在 \mathcal{NPC} 或者 \mathcal{P} 中的语言，其中字母 J 表示中间的。这种上下文中，词中间的的最好解释是通过解决判定问题体现出来的。一个 \mathcal{NPJ} 中的问题不是 \mathcal{NP} -难的，因此也不被看作跟 \mathcal{NPC} 中的问题那样难。另一方面，由于它不在 \mathcal{P} 中，因此我们认为它比 \mathcal{P} 类中的问题更难。词语中间的来自于这样一种解释，即 \mathcal{NPJ} 中的问题比 \mathcal{P} 类中的问题难但是又不像 \mathcal{NPC} 中的问题那么难。

如果 $\mathcal{P} = \mathcal{NP}$ ，那么 \mathcal{NPJ} 类为空。定理 17.1.1 没有给出证明，它保证了如果 $\mathcal{P} \neq \mathcal{NP}$ ，那么中间问题是存在的。

定理 17.1.1 如果 $\mathcal{P} \neq \mathcal{NP}$ ，那么 \mathcal{NPJ} 类非空。

我们把字母表 Σ 上的语言 L 的补表示为 \bar{L} ，它包含所有不在 L 中的字符串；也就是说 $\bar{L} = \Sigma^* - L$ 。一个语言家族 \mathcal{F} 是封闭的，如果当 $L \in \mathcal{F}$ 时 $\bar{L} \in \mathcal{F}$ 。家族 \mathcal{P} 对于补运算是封闭的。一个可以在多项式时间内接收语言的确定型图灵机可以被转换成在同样的多项式边界中接收其补集。这个转换仅仅包括交换图灵机的接收状态和拒绝状态。

不确定性的不对称性对接收语言的机器以及接收其补的机器的复杂性有重大影响。为了得到肯定答案，我们所需要的仅是一个可以验证肯定答案的不确定的“猜想”。仅当所有猜想失败时，我们才会得到一个否定的答案。我们用可满足性问题来说明一个语言及其补的不确定型接收的复杂性的不对称性。

可满足性问题的输入是布尔变量集 V 上的一个合取范式公式 u ，如果 u 是可满足的那么输出为是，否则输出否。定理 15.5.2 描述了一个在多项式时间解决可满足性问题的非确定型机器。它通过猜测 V 上的真值赋值来完成计算。检查是否一个真值赋值满足公式 u 是一个直接可以在 u 长度的多项式时间内完成的过程。

可满足性问题的补是指判断是否一个合取范式公式是不可满足的；也就是说，它不能被任意的真值赋值满足。如果 u 不能被所有可能的真值赋值满足，那么我们可以得到肯定的答案。解决“不可满

530

足性问题”的一个非确定型策略要求一个能够验证 u 的不可满足性的猜想。这个猜想不能只是真值赋值，因为发现一个不满足 u 的真值赋值不足以说明 u 是不可满足的。直觉上，我们需要知道所有可能的真值赋值时 u 的真值。如果 $\text{card}(V) = n$ ，那么我们需要检查 2^n 个真值赋值。因此，说这个问题在 NP 内似乎是合理的。注意我们在上一个句子中使用了词语直觉上和似乎是。这么说是因为不可满足性问题是否在 NP 中仍然是未知的。

除了考虑可满足性问题的补，我们还要检查所有 NP 中的语言的补所组成的语言族。这个族是 $\text{co-NP} = \{\bar{L} \mid L \in \text{NP}\}$ 。

定理 17.1.2 如果 $\text{NP} \neq \text{co-NP}$ ，那么 $\mathcal{P} \neq \text{NP}$ 。

证明：如前所述， \mathcal{P} 在补运算上是封闭的。如果 NP 对于补运算不封闭，那么这两个语言类不能相同。

定理 17.1.2 为回答 $\mathcal{P} = \text{NP}$ 问题提供了另外一种方法。找到一种语言 $L \in \text{NP}$ 且 $L \notin \text{co-NP}$ 就足够了。 $\text{NP} = \text{co-NP}$ 的证明不能回答 \mathcal{P} 和 NP 是否相同的问题。此时，是否 $\text{NP} = \text{co-NP}$ 仍然是未知的。正如大家通常所相信的 $\mathcal{P} \neq \text{NP}$ ， $\text{NP} \neq \text{co-NP}$ 也是理论计算机科学家的一致意见。然而，大多数人的意见不能判定数学属性，人们仍然在为了证明这些不等式而努力。定理 17.1.3 为说明 NP 和 co-NP 的相等关系提供了一种方法。

定理 17.1.3 如果存在一个 NP -完全语言 L 使得 $\bar{L} \in \text{NP}$ ，那么 $\text{NP} = \text{co-NP}$ 。

证明：假设 L 是满足上述条件的语言。我们首先要说明的是，在这些条件下，任意一个 NP 中的语言 Q 的补也在 NP 中。因为 L 是 NP -完全的，所以存在一个从 Q 到 L 的多项式时间归约。这个归约同样也是 \bar{Q} 到 \bar{L} 的归约。

根据我们的假设 $\bar{L} \in \text{NP}$ ，一个非确定型图灵机可以在多项式时间接收 \bar{L} 。结合执行从 \bar{Q} 到 \bar{L} 的归约的机器和接收 \bar{L} 的机器，我们可以得到一个能在多项式时间内接收 \bar{Q} 的非确定型机器。因此， $\text{co-NP} \subseteq \text{NP}$ 。

为了证明 $\text{NP} = \text{co-NP}$ ，我们必须证明反向的结论。设 Q 是 NP 中的任意一个语言。根据上述证明， \bar{Q} 也在 NP 中。所以 \bar{Q} 的补，也就是 Q ，在 co-NP 中。

我们可以使用可满足性问题和它的补来初始化 co-NP 族的检查。我们曾经说相信可满足性问题的补不在 NP 中似乎是合理的。根据定理 17.1.2， \bar{L}_{SAT} 在 NP 中，当且仅当 $\text{NP} = \text{co-NP}$ 。图 17-1 表示了我们推测的 \mathcal{P} 、 NP 、 NPC 和 co-NP 之间的关系。

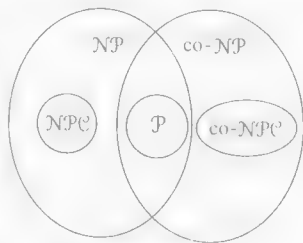


图 17-1 $\mathcal{P} \neq \text{NP}$ 且 $\text{NP} \neq \text{co-NP}$ 时的包含关系

531

17.2 空间复杂性

前面几章都关注于图灵机和判定问题的时间复杂性。我们也可以同样的选择去分析计算所要求的空间。在高层算法问题求解中，程序所要求的时间和存储通常是相关的。我们将说明一个图灵机的时间复杂性提供了所需空间的上界，反之亦然。除非其他说明，我们所介绍的空间复杂性对于确定型和非确定型图灵机都成立。17.3 节将介绍在接收语言时，我们为了限制一个计算可得的空间所做的努力。

图 17-2 显示的图灵机结构用于度量一个计算所需要的空间。带 1 包括输入且是只读的。输入长度为 n 的字符串时，输入带的带头必须保持在位置 0 和 $n+1$ 之间。图灵机可以读输入带但是在其他带上执行操作。提供一个输入带可以分离输入需要的空间和计算所需要的空间。有时我们把满足前述条件的图灵机叫做离线图灵机（off-line turing machine），因为输入可以看作是被离线地优先于计算提供的，并且它不包括在资源使用的评估中。除非其他说明，我们假设本章后续部分中的图灵机都是针对空间复杂性分析而设计的。

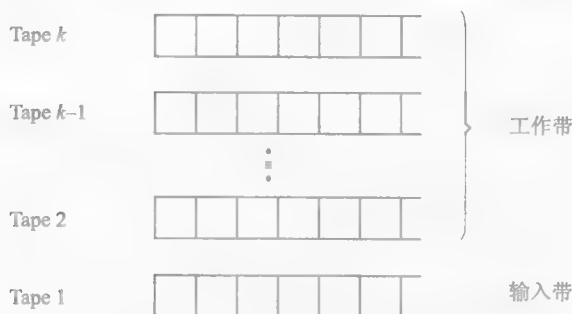


图 17-2 空间复杂性的图灵机结构

定义 17.2.1 一个 $k+1$ 带图灵机 M 的空间复杂性 (space complexity) 是一个函数 $sc_M: N \rightarrow N$, 当图灵机 M 被一个长度为 n 的字符串初始化时, $sc_M(n)$ 是 M 的计算从其任意工作带上读的带方格的最大值。

这个定义对于确定型和非确定型图灵机都适用。对于非确定型机器, 每个长度为 n 的输入字符串的可能的计算上都会出现这个最大值。与时间复杂性不同, 我们不会假设图灵机的计算对每个输入都停机, 即使计算永不终止, 机器的带头仍然可以保持在带上初始片段的有限长度内。

532

空间复杂性永远大于 0。即使一个图灵机不发生任何转换, 机器也必须读工作带的最左端位置以便作此决定。由于空间复杂性只度量工作带, 因此 $sc_M(n) < n$ 是可能的。也就是说, 计算所需要的空间可能小于输入的长度。在例 17.2.1 中, 我们设计了另外一个接收回文的机器来说明一个空间复杂性为 $O(\log_2(n))$ 的计算。

例 17.2.1 例 14.3.1 构造了一个接收 $\{a, b\}$ 上的回文的双带图灵机。这个机器遵循了为分析空间复杂性而设计的机器的描述。输入带是只读的并且带头只读输入字符串和输入另一端的空白。 M 的空间复杂性是 $n+2$; 计算在带 2 上重新生成输入并且反向地读入字符串的方式来比较带 1 和带 2 上的字符串。

现在, 我们设计一个空间复杂性 $sc_M(n) = O(\log_2(n))$ 的接收回文的三带机器 M , 工作带用作计数器和记录自然数的二进制表示。我们的策略是, 使用计数器识别和比较字符串的第 i 个元素和右边第 i 个元素。如果它们匹配, 那么计数器增加并且比较第 $i+1$ 个元素。这个过程持续到发现一组不匹配的元素或者所有元素都被比较过了。在前一种情况下, 字符串被拒绝, 而在后一种情况下字符串被接收。

输入 u 的长度为 n 时, M 的一个计算包括如下步骤:

1. 在带 3 的位置 1 写入 1。
2. 将带 3 拷贝到带 2。
3. 将输入带的带头放置在最左边的方格。设 i 是带 2 和带 3 上的二进制表示所代表的整数。
4. 当带 2 上的数字不是 0 时,
 - a) 把输入带的带头向右移动一格。
 - b) 减小带 2 上的值。
5. 如果输入带上读入的字符为空, 停机并接收字符串。
6. 使用机器状态记录输入的第 i 个字符。
7. 把输入带的带头移动到紧邻输入右侧的位置 (带位置 $n+1$)。
8. 把带 3 拷贝到带 2。
9. 当带 2 上的字符不是 0 时,
 - a) 把输入带的带头向左移动一格。
 - b) 减小带 2 上的值。
10. 如果第 $(n-i+1)$ 个字符与第 i 个字符匹配, 那么增加带 3 上的值, 带头返回到它们的初始位

533

置,并且计算回到步骤2继续。否则计算拒绝该输入。

带2和带3上的操作可以增加和减小一个自然数的二进制表示。因为 $n+1$ 是写入到这些带上的最大值,因此每个带至多使用 $\lceil \log_2(n+1) \rceil + 2$ 个带上的方格。□

我们把一个离线图灵机叫做 $s(n)$ 空间边界的(space-bounded),如果输入长度为 n 时计算的工作带所使用的带上方格至多为 $\max\{1, s(n)\}$ 。空间复杂性函数 $sc_M(n)$ 描述了输入 n 时 M 的计算实际所需要的最大空间,其中可能没有达到空间边界提供的上界。正如前面关于空间复杂性的描述,即使一个图灵机的计算不会终止,它仍然可以是空间边界的。

我们可以使用一个只有一条带的 $s(n)$ 空间边界的机器来模拟空间边界 $s(n) \geq n$ 的 $k+1$ 带图灵机 M 。这与时间复杂性的度量不同,带数量的归约会导致时间复杂性的增加。这个证明使用了8.6节介绍的从一个 k 带图灵机到 $2k+1$ 道图灵机的构造。结果,多道图灵机所扫描的带方格的数量正好是原多带图灵机的任何一个工作带读入的最大值。下面的定理对上述内容进行了总结。

定理 17.2.2 设 L 是一个能被空间边界 $s(n) \geq n$ 的 $k+1$ 带图灵机 M 接收的语言。那么 L 也可以被一个 $s(n)$ 空间边界的单工作带图灵机接收。

[534]

正如定理17.2.2,我们会很频繁地使用到图灵机空间复杂性 $sc_M(n) \geq n$ 或者空间边界 $s(n) \geq n$ 的假设。在定理中增加这些条件是为了保证至少 n 个带方格可得。由于空间复杂性函数本身就是一个空间边界,因此第一个条件蕴含第二个条件。反之不成立。例17.2.1描述的图灵机 M 是 $s(n) = n+2$ 空间边界的,但是它的空间复杂性不满足 $sc_M(n) \geq n$ 。

尽管我们对空间复杂性的定义是基于多带离线图灵机的计算的,但是空间边界的表示对于单带图灵机同样适用。如果一个单带图灵机使用的带方格的最大数量至多为 $\max\{n+1, s(n)\}$,那么这个单带图灵机是 $s(n)$ 空间边界的。当仅有一条带时,存储输入所需要的空间也包含在边界中。

根据一个机器是 $s(n) \geq n$ 空间边界的假设和定理17.2.2,方便时我们可以把注意力限制在一条工作带上。事实上,在空间边界 $s(n) \geq n$ 时任何能够被接受的语言都能被一个满足同样空间边界的单带确定型图灵机接受(练习9)。这个证明使用的归约与从多道机器到单道机器的归约相同。

在研究空间复杂性时选择离线图灵机的原因是为了采纳一个适用于所有空间边界分析的图灵机模型。许多有趣的语言可以被空间边界小于输入长度的机器所接收。特别地,人们广泛地研究了可以被 $\log_2(n)$ 空间边界图灵机接收的语言类。然而,我们的注意力集中于这样的问题,它们可能需要大量的资源,并且对于这些问题而言,输入可用的空间至少等于输入规模是合理的。

17.3 空间复杂性和时间复杂性的关系

我们可以使用图灵机的时间复杂性来获取空间复杂性的上界。一个带头在一个计算中可读的带方格的数量受制于计算中转换的次数。

定理 17.3.1 设 M 是一个时间复杂性为 $tc_M(n) = f(n)$ 的 $k+1$ 带图灵机。那么 $sc_M(n) \leq f(n) + 1$ 。

证明: 当 M 的每一个转换都把工作带的带头右移时,计算使用的带的数量最大。在这种情况下,从工作带读入的最大带方格数量是 $f(n) + 1$ 。■

由于机器可以多次读入带的一个特定段,因此根据已知的空间边界来获取时间复杂性约束更加复杂。我们使用一个双带图灵机 M 来说明可以根据机器的空间复杂性得到一个确定型图灵机的计算的时间边界。我们假设 M 对所有输入字符串都停机,这是时间复杂性的要求。从双带机器到 $k+1$ 带机器的泛化是简单的。

[535]

定理 17.3.2 设 M 是一个可以对任意输入字符串停机的双带图灵机, M 的空间边界是 $s(n)$ 。那么 $tc_M(n) \leq m \cdot s(n) \cdot (n+2) \cdot t^{(n)}$,其中 m 是 M 的状态的数量, t 是 M 的带字符的数量。

证明: 设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个双带图灵机, $m = \text{card}(Q)$, $t = \text{card}(\Gamma)$ 。对于每个长度为 n 的输入,空间边界把 M 的计算约束为至多是带2上的 $s(n)$ 位置。通过把计算限制在带上有限长度的片断,我们能够计算 M 所进入的不同机器配置的数量。

工作带的每一个位置可以含有 t 个字符中的任意一个,这导致了 $t^{(n)}$ 个可能的配置。带1的带头可以任意地读前 $n+2$ 个位置,带2的带头可以从位置0到位置 $s(n)-1$ 读入。因此,一共有 $s(n) \cdot$

$(n+2) \cdot t^{s(n)}$ 个可能的机器配置和带头位置的组合。对于任意一个这样的组合，机器可以处于 m 个状态中的一个，这样就产成了 $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ 个不同的配置。

确定型机器的配置的重复说明机器已经进入了一个无限循环。因为 M 对所有计算停机，所以计算必须在达到 $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ 次转换前停机。 ■

对于一个非确定型机器，一个终止的计算可以有多个可能的配置的数目转换次数。当一个配置被重复时，计算可以选择一个不同的转换。在推论 17.3.3 中，我们使用配置数据的限制来为任何一个空间边界图灵机接收字符串所需要的转换次数产生一个指数级的边界。边界以指数形式给出，化简了下一个部分中确定型和非确定型计算所需要的空间数量的比较。根据定理 17.2.2，考虑只有一个工作带的图灵机就已经足够了。

推论 17.3.3 设 M 是一个空间边界 $s(n) \geq n$ 的图灵机。存在一个依赖于 M 的状态数量和带符号数量的常数 c ，使得任意可以被 M 接收的长度为 n 的字符串也可以被一个至多有 $c^{s(n)}$ 次转换的计算接收。

证明：我们再次设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个双带图灵机， $m = \text{card}(Q)$ ， $t = \text{card}(\Gamma)$ 。根据定理 17.3.2 的证明，输入长度为 n 时，任何一个计算都存在 $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ 个可能的配置。机器配置数量的指数级边界的获得使用下面的不等式：

$$(n+2)s(n) \leq 3^{s(n)},$$

当 $n \leq s(n)$ 且 $s(n) > 0$ 时该不等式成立。我们可以通过把 $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ 的项替换成把 $s(n)$ 当指数的函数来获得转换次数的边界：

[536]

$$\begin{aligned} m \cdot s(n) \cdot (n+2) \cdot t^{s(n)} &\leq m^{s(n)} \cdot s(n) \cdot (n+2) \cdot t^{s(n)} \\ &\leq m^{s(n)} \cdot 3^{s(n)} \cdot t^{s(n)} \\ &= (3mt)^{s(n)} \\ &= c^{s(n)} \end{aligned}$$

c 可以直接根据图灵机 M 的状态个数和带符号个数获得。

M 的任何一个多于 $c^{s(n)}$ 次转换的计算都必须重复一个配置。一个接收字符串 w 的这种形式的转换可以写作

$$\begin{aligned} q_0: & \cdot BwB, \cdot BB \\ \vdash q_1: & Bu \cdot vB, x \cdot y \\ \vdash q_i: & Bu \cdot vB, x \cdot y \\ \vdash q_j: & Bu' \cdot v'B, x' \cdot y', \end{aligned}$$

其中，第一个分号后的字符串表示带 1，第二个字符串表示带 2，点表示带头正在读右边的字符。从计算中删除重复配置的部分可以得到另一个长度更小的接收计算：

$$\begin{aligned} q_0: & \cdot BwB, \cdot BB \\ \vdash q_1: & Bu \cdot vB, x \cdot y \\ \vdash q_j: & Bu' \cdot v'B, x' \cdot y'. \end{aligned}$$

这个过程可以一直重复直到产生一个长度小于 $c^{s(n)}$ 的计算。 ■

图灵机 M 接收字符串所需的转换次数的上界可以用来构造一个能够接收同样语言、有同样空间复杂性并对所有输入字符串停机的机器。其基本思想是给 M 添加一个带用于记录转换的次数。我们把计数带初始化为推论 17.3.3 提供的边界。每个发生一次转换，计数器减 1。如果计数器的值为 0，那么计算停止并拒绝该输入。这个构造中，我们惟一关心的是要保证技术带使用的带不超出 M 的空间边界允许的范围。我们可以通过选择一个合适的底 b 并把技术带上的数字表示成以 b 为底的系统来做到这一点。

推论 17.3.4 设 L 是一个可以被图灵机在空间边界 $s(n) \geq n$ 内接收的语言。那么 L 可以被一个空间边界为 $s(n)$ 且对任何输入都停机的图灵机 M' 接收。

[537]

一个空间边界 $s(n)$ 是完全空间可构造的 (full space constructible), 如果存在一个图灵机 M , 并且其每一个长度为 n 的字符串的计算正好访问 $s(n)$ 个带方格。如果 M 是一个空间复杂性为 $sc_M(n) = s(n) \geq n$ 的图灵机, 那么 $s(n)$ 是完全空间可构造的 (练习 5)。完全空间可构造函数集的集合包括 n' , 2^n , $n!$ 和多数普通的数论函数。此外, 如果 $s_1(n)$ 和 $s_2(n)$ 都是完全空间可构造函数, 那么 $s_1(n)s_2(n)$, $2^{s_1(n)}$ 和 $s_2(n)^{s_1(n)}$ 也都是完全空间可构造函数。以上发现使我们得到一个结论, 即任何把 2 做为指数链的函数

$$s(n) = 2^{2^{\cdot^{2^n}}}$$

是完全空间可构造函数。因此, 图灵机计算所需要的空间数量没有限制。定理 17.3.5 列出了一些条件, 在这些条件下, 增加计算可用的空间会导致可以接收的语言族的增加。

定理 17.3.5 设 $s_1(n) \geq n$ 和 $s_2(n) \geq n$ 是从 N 到 N 的函数, 使得

$$\inf_{n \rightarrow \infty} \frac{s_1(n)}{s_2(n)} = 0$$

并且 s_2 是完全空间可构造的, 那么存在一个可以被 $s_2(n)$ 空间边界的图灵机接收的语言 L , 而 L 不能被任何 $s_1(n)$ 空间边界的图灵机接收。

证明: 我们构造一个五带 $s_2(n)$ 空间边界的图灵机 M , 并使其语言不能被任何 $s_1(n)$ 空间边界的图灵机接收。 M 的输入是 $\{0,1\}^*$ 上的字符串, 并且计算把这样一个字符串的解释用作一个双带图灵机。当运行输入字符串 w 时, M 的计算包含了两个图灵机计算的模拟。第一个配置 M 的一个带以保证 $s_2(n)$ 空间边界, 第二个配置模拟给字符串 w 编码的机器的计算。当运行字符串 w 时, 我们称之为 M_w 。我们给出一个对角化证明来说明 M 的语言不被 $s_1(n)$ 空间边界的图灵机接收。

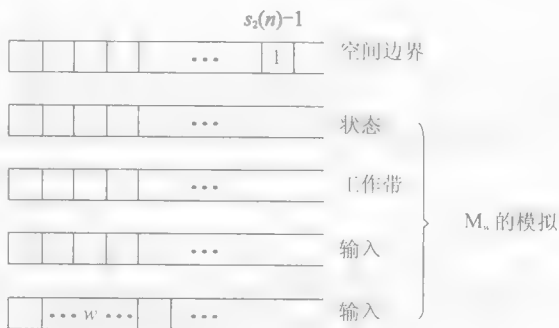
我们使用 14.6 节介绍的多带图灵机的编码, 但是我们允许在开始编码时的 000 之前有任意多个 1。因此, 如果 $w \in \{0,1\}^*$ 是图灵机的编码, 那么字符串 $1w, 11w, 111w, \dots$ 是同一个机器的编码。通过这个改动, $\{0,1\}^*$ 字符串的枚举包括了每个图灵机的无限个编码。任何双带图灵机编码要求的字符串 w 都可以被看作是双带、单状态且无转换的机器。

输入为 w 时, M 的计算开始于把带 5 上的第 $s_2(n) - 1$ 个位置标记为 1 (因为 $s_2(n)$ 是完全空间可构造的, 所以存在这样一个图灵机, 当它运行输入 w 时正好使用了 $s_2(n)$ 个方格。输入为 w 时这个机器的计算可以在带 2 和带 4 上模拟, 并且计算中被访问的最右边的方格被记录在带 5 上。

在说明了带边界后, M 在带 2 和带 4 上模拟输入为 w 时机器 M_w 的计算。在这个阶段开始时, M 如下图所示:

在模拟 M_w 的过程中, 带 3 和带 5 的带头同步地移动。如果带头试图移动到带 5 上最右边的标记, 那么 M 的计算停机且拒绝该输入。这样能够保证 M 是 $s_2(n)$ 空间边界的。如果计算达到了空间边界并没有终止并且 M_w 拒绝 w 停机, 那么机器 M 接收输入字符串 w 。

现在我们来证明 $L(M)$ 不能被任何 $s_1(n)$ 空间边界的图灵机 M' 接收。根据推论 17.3.4, 我们可以假设 M' 对所有输入停机。 M' 的编码会导致出现 $\{0,1\}^*$ 枚举的无限多项。因为



$$\inf_{n \rightarrow \infty} \frac{s_1(n)}{s_2(n)} = 0,$$

所以存在某个 $n \geq \text{length}(w)$ 使得 $s_1(n) < s_2(n)$ 。我们可以通过在字符串 w 开头追加 1 来生成一个长度正好为 n 的 M' 的 w' 的编码。

现在我们考虑运行输入 w' 时机器 M 的计算。由于 $s_1(n) < s_2(n)$, 所以 M 有足够的空间来模拟 M' 的计算。因此 M 接收 w' , 当且仅当 M' 不接收 w' 。因此, $L(M) \neq L(M')$ 。据此可得不存在能够接收 $L(M)$ 的 $s_1(n)$ 空间边界的图灵机。 ■

2^n 和 2^{2^n} 的空间可构造性结合定理 17.3.5 可以保证存在这样的语言 L , L 不能被任何空间边界为 2^n 的机器接收。后面的边界比任何多项式的增长速度都快。因此不存在能够接收 L 的多项式空间复杂性图灵机。由于空间复杂性为时间复杂性提供了一个边界, 所以 L 不能被在多项式时间接收, 故而语言 L 的成员资格问题是难解的。 [539]

在没有确定一种空间和时间复杂性不是多项式边界的特定语言的情况下, 上述的证明说明了难解语言的存在性。我们将在 17.5 节说明, 关注于使用正则表达式描述的语言的问题需要指数级的空间。

17.4 \mathcal{P} -空间, \mathcal{NP} -空间和萨维奇定理

类 \mathcal{P} 和类 \mathcal{NP} 分别包含了能被确定型和非确定型图灵机在多项式时间内接收的语言。我们可以使用一种类似的方式定义可以被这样的图灵机接收的语言, 这些图灵机的计算所需要的空间数量仅随输入长度多项式地增长。

定义 17.4.1 一个语言 L 是多项式空间可判定的 (decidable in polynomial space), 如果存在一个 $sc_M \in O(n^r)$ 的且接收 L 的图灵机 M , 其中 r 是一个独立于 n 的自然数。确定型图灵机多项式时间可判定的语言族记做 \mathcal{P} -空间。类似地, 非确定型图灵机多项式时间可判定的语言族记做 \mathcal{NP} -空间。

关于这些新的复杂性类, 有一些明显的结论。显然, \mathcal{P} -空间 $\subseteq \mathcal{NP}$ -空间。此外, 根据定理 17.3.1, $\mathcal{P} \subseteq \mathcal{P}$ -空间且 $\mathcal{NP} \subseteq \mathcal{NP}$ -空间。自 20 世纪 60 年代被提出以来, \mathcal{P} 是否是 \mathcal{NP} 的一个子集是一个尚未解决的公开的问题。类似的空间复杂性问题的答案是已知的, 即 \mathcal{P} -空间 = \mathcal{NP} -空间。时间和空间复杂性的本质区别是空间可以在计算中被复用。

我们将说明, 每一个能被非确定型 $s(n)$ 空间边界的图灵机接收的语言都能确定地在空间边界 $O(s(n)^2)$ 内被接收。因此我们立即可以得知, 一个能被非确定型图灵机在多项式空间内接收的语言也能够被确定型图灵机在多项式空间内接收。像往常一样, 我们将把我们的注意力集中在双带图灵机上。

根据非确定型图灵机构造一个等价的确定型图灵机, 必须描述一个能够系统地检查非确定型机器的所有备用计算的方法。首先我们考虑输入为 w 时, 构造非确定型图灵机的备用计算的标准方法所需要的空间。在这种方法中, 空间分析的一个严格的特性是需要存储当前计算中能够生成成功计算的所有机器配置。我们使用一个栈来维护和访问这个配置, 从而产生非确定型计算的深度优先分析。 [540]

设 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 是一个空间边界为 $s(n)$ 的双带图灵机。输入为 w 时 M 的计算形式如下:

$$\begin{aligned} q_0 &: \cdot BwB \cdot BB \\ \vdash q_i &: Bu \cdot vB, x \cdot y \\ \vdash q_j &: Bu' \cdot v'B, x' \cdot y'. \end{aligned}$$

如果从机器配置 $q_j: Bu' \cdot v'B, x' \cdot y'$ 没有转换且 q_j 不是接收状态, 或者所有可运行的转换都已经检查完毕, 那么计算必须回到 $q_i: Bu \cdot vB, x \cdot y$ 以尝试备用的转换。一个机器配置栈提供了测试所有备用计算所需要的后进先出策略。为了判断这个策略的时间复杂性, 我们必须回答两个问题: “表示机器配置需要多少空间?” 和 “可能存储的配置的最大数量是多少?”。

一个空间边界为 $s(n)$ 的双带图灵机的配置表示需要对所有的机器状态、只读带的带头位置、工作带的带头位置以及工作带的前 $s(n)$ 个带方格进行编码。对于每个长度为 n 的输入, 状态所需要 $\lceil \log_2(\text{card}(Q)) \rceil$ 个格, 输入带带头位置所需要 $\lceil \log_2(n+2) \rceil$ 个格, 工作带带头位置需要个 $\lceil \log_2 s(n) \rceil$ 格, 工作带需要 $s(n)$ 个格。因此整个配置可以在 $O(s(n))$ 空间内被编码。

第二个问题的答案说明这种把非确定型机器转换为确定型机器的直接方法不会产生期望的确定型机器的空间复杂性。根据定理 17.3.2, 需要存储在栈中的配置的数量随 $s(n)$ 指数级地增长。因此我们需要其他方法。

重要发现, 即高效复用空间是一个有 k 次转换的计算

$$\begin{aligned} q_0 &: \cdot BwB \cdot BB \\ \vdash q_j &: Bu \cdot vB, x \cdot y, \end{aligned}$$

可以被分裂成两个计算

$$q_0: BwB, .BB$$

$$\vdash q_j: Bu'.v'B, x'.y'$$

$$\vdash q_i: Bu.vB, x.y,$$

每个计算均包含 $k/2$ 次转换, 如果顺序地执行这两个计算, 那么第一个计算使用的空间可以被第二个计算使用。

541

我们使用这种存储复用策略来判断是否一个字符串 w 被空间边界为 $s(n)$ 的双带非确定型图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 接收。设 cf_1, cf_2, \dots, cf_p 是 w 在带上时, 所有可能的机器配置的列表, 其中 cf_1 是初始配置 $q_0: .BwB, .BB$ 的编码。空间边界 $s(n)$ 保证了配置的数量是有限的 (定理 17.3.2)。

算法使用分治法来判断一个配置 cf_i 是否是通过 k 或更少次转换从配置 cf_1 得到的。为了回答这个问题, 我们只需找到一个配置 cf_{i_1} , 使得

1. 在 $k/2$ 或更少次转换内 $cf_{i_1} \vdash cf_{i_1}$, 且
2. 在 $k/2$ 或更少次转换内 $cf_{i_1} \vdash cf_i$ 。

类似地, 为了发现是否在 $k/2$ 或更少次转换内有 $cf_1 \vdash cf_i$, 我们只需找到一个配置 cf_{i_1} , 使得

1. 在 $k/4$ 或更少次转换内 $cf_{i_1} \vdash cf_{i_1}$, 且
2. 在 $k/4$ 或更少次转换内 $cf_{i_1} \vdash cf_i$ 。

算法 17.4.4 中的 *Derive* 过程使用递归来执行这个搜索。图 17-3 给出了与调用 $Derive(cf_1, cf_i, k)$ 相关的递归树。节点 $[m, n]$ 表示一个判断 cf_n 是否是由 cf_m 推导而来的调用。如图 17-3 所示, $Derive(cf_1, cf_i, k)$ 的评估最多有 $\lceil \log_2(k) \rceil$ 次嵌套的递归。现在我们可以用以上的发现来为接收非确定型机器定义的语言的确定型算法生成一个空间边界。

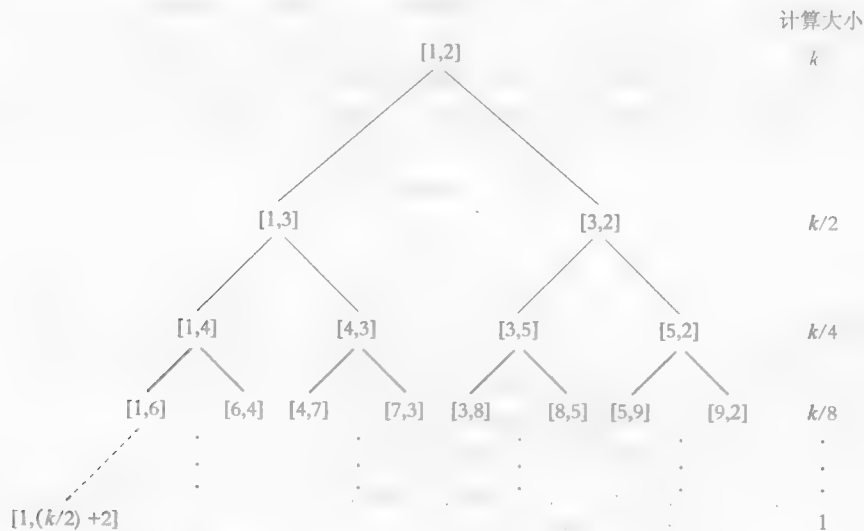


图 17-3 $Derive(cf_1, cf_2, k)$ 的递归树

定理 17.4.2 (萨维奇 (Savitch) 定理) 设 M 是一个双带非确定型图灵机, 其空间边界为 $s(n)$ 。那么 $L(M)$ 可以被空间边界为 $O(s(n)^2)$ 的确定型图灵机接收。

证明: 算法 17.4.4 描述了字符串 w 的推导的递归查找。根据推论 17.3.3, 每一个 $w \in L(M)$ 都能被一个转换次数不超过 $c^{|w|}$ 的计算接收。我们顺序地检查机器配置, 并且对于每个 M 的接收配置都在步骤 3.2 中调用递归查找过程 *Derive*。该调用的参数是 M 的初始配置, 一个接收配置和转换边界 $c^{|w|}$ 。如果某个 *Derive* 调用发现了一个推导, 那么算法停止并接收该字符串。如果所有的调用都失

败, 那么 w 是不可推导的并且字符串被拒绝。

接下来我们要判断该方法需要的存储数量。在一个递归调用中, 调用过程 *Derive* 存储一个包含了它的变量值和局部变量在内的活动记录 (activation record)。当调用结束时, 活动记录可用于重新存储值。*Derive* 的活动记录包括两个机器配置和表示转换边界的整数。

一个实现该算法的图灵机必须在带上存储活动记录。如前所述, 一个机器配置只需要 $O(s(n))$ 个带方格, 因此活动记录所需要的空间也是 $O(s(n))$ 。嵌套调用的最大次数是

$$\lceil \log_2(c^{s(n)}) \rceil = \lceil s(n) \log_2(c) \rceil \in O(s(n)).$$

这样, $O(s(n))$ 个活动记录的总空间是 $O(s(n)^2)$ 。

定理 17.4.2 中的空间复杂性边界可用于说明 \mathcal{P} -空间 = \mathcal{NP} -空间。

推论 17.4.3 如果 L 在 \mathcal{NP} -空间, 那么 L 在 \mathcal{P} -空间。

证明: 如果 L 在 \mathcal{NP} -空间, 那么 L 可以被多项式空间边界 $p(n)$ 的非确定型图灵机接收。根据定理 17.4.2, L 可以被多项式空间边界 $O(p(n)^2)$ 的确定型图灵机接收。因此 L 在 \mathcal{P} -空间。

算法 17.4.4

非确定型图灵机的递归模拟

输入: 图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$

字符串 $w \in \Sigma^*$

M 的配置 cf_1, cf_2, \dots, cf_p

常量 $c = 3 \cdot \text{card}(Q) \cdot \text{card}(\Gamma)$

空间边界 $s(n)$

1. $\text{found} = \text{false}$

2. $i = 1$

3. **while** not found and $i < p$ **do** (检查所有的接收配置)

3.1. $i := i + 1$

3.2. cf_i 是 M 的一个接收配置

then $\text{found} = \text{Derive}(cf_1, cf_i, c^{s(n)})$

end while

4. **if** found **then** 接收 **else** 拒绝

$\text{Derive}(cfs, cfe, k);$

begin

$\text{Derive} = \text{false}$

if $k = 0$ and $cfs = cfe$ **then** $\text{Derive} = \text{true}$

if $k = 1$ and $cfs \vdash cfe$ **then** $\text{Derive} = \text{true}$

if $k > 1$ **then do**

$i = 1$

while not Derive and $i < p$ **do** (检查所有中间配置)

$i := i + 1$

$\text{Derive} = \text{Derive}(cfs, cf_i, \lceil k/2 \rceil)$ **and** $\text{Derive}(cf_i, cfe, \lfloor k/2 \rfloor)$

end while

end if

end.

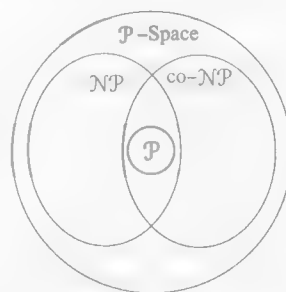


图 17-4 \mathcal{P} -空间与其他复杂性类的关系

图 17-4 说明了 \mathcal{P} -空间与其他复杂性类的关系。 \mathcal{P} -空间 = \mathcal{NP} 和 \mathcal{P} -空间 = \mathcal{P} 是否成立仍然是未知的。然而, 我们相信图 17-4 的所有结论是合适的。

17.5 \mathcal{P} -空间完全性

[544]

我们引入 \mathcal{P} -空间完全性的表示来描述 \mathcal{P} -空间类的一般性问题, 并提供一种判断 $\mathcal{P} \subseteq \mathcal{P}$ -空间和 $\mathcal{NP} \subseteq \mathcal{NP}$ -空间中哪个结论是等式 (如果有的话) 的方法。

定义 17.5.1 我们把一个语言 Q 叫做 \mathcal{P} -空间难的 (\mathcal{P} -space hard), 如果对于每个 $L \in \mathcal{P}$ -空间, L 都可以在多项式时间归约到 Q 。一个在 \mathcal{P} -空间内的 \mathcal{P} -空间难的语言叫做 \mathcal{P} -空间完全的 (\mathcal{P} -space complete)。

注意 \mathcal{P} -空间完全性定义中的归约有多项式的时间, 而不是空间约束。这个要求保证了发现 \mathcal{P} -空间完全问题的多项式时间解蕴含 \mathcal{P} -空间 = \mathcal{P} 。

定理 17.5.2 设 Q 是 \mathcal{P} -空间完全的语言, 那么

- i) 如果 Q 在 \mathcal{P} 内, 那么 \mathcal{P} -空间 = \mathcal{P} 。
- ii) 如果 Q 在 \mathcal{NP} 内, 那么 \mathcal{P} -空间 = \mathcal{NP} 。

我们可以根据所有在 \mathcal{P} -空间内的语言到 \mathcal{P} -空间完全语言的可归约性, 以及我们所熟悉的通过顺序执行两个多项式时间边界的机器获取多项式时间边界的过程, 来证明定理 17.5.2。定理 17.5.2 说明, 找到一个不在 \mathcal{P} 或 \mathcal{NP} 内的 \mathcal{P} -空间完全语言就能够回答这些类是否在 \mathcal{P} -空间内的问题。

本节的后续部分主要用于说明下面定义的判定问题是 \mathcal{P} -空间完全的。

输入: 字母表 Σ 上的正则表达式 α

输出: 是; 如果 $\alpha \neq \Sigma^*$

否; 否则

为了证明这个问题是 \mathcal{P} -空间完全的, 我们需要两个步骤。首先, 我们必须为正则表达式和在多项式空间内解决该问题的图灵机设计一个字符串表示。也就是说, 图灵机接收一个字符串, 当且仅当它是这样一个正则表达式, 其语言不包括其字母表上的所有字符串。这个步骤可以在字符串接收的层次完成, 我们把它留作练习。如果一个语言包括了不能描述所有字符串的正则表达式, 那么我们把它记做 L_{REG} 。

第二个步骤要说明 \mathcal{P} -空间中的所有语言都能在多项式时间归约到 L_{REG} 。该证明需要使用到那些在可满足性问题 \mathcal{NP} 完全性证明中使用的策略。为了说明 \mathcal{P} -空间的语言 L 能够归约到 L_{REG} , 我们把接收 L 的空间边界的图灵机 M 的计算转换成正则表达式。对于每个 $w \in \Sigma_M^*$, 我们构造一个正则表达式 α_w 使得 M 接收 w , 当且仅当 α_w 不包括其字母表上的所有字符串。

[545]

设 $M = (Q, \Sigma_M, \Gamma_M, \delta, q_0, F)$ 是一个空间边界为 $s(n)$ 的单带确定型图灵机。我们给 M 的字母表添加下表以便把它与正则表达式的字母表区分开来。我们将根据 M 和 w 来构造正则表达式。在不损失一般性的前提下, 我们假设没有始于 M 的接收状态的转移。

首先, 我们定义一个字母表 Σ , 该字母表允许我们把 M 的计算表示成 Σ 上的字符串。字母表对于每个 $q_i \in Q$ 和 $a \in \Gamma_M$ 都包含 $[q_i, a]$ 和 $[*, a]$ 形式的有序对。除了有序对, Σ 还包含字符 \vdash 。直观上, 一个有序对 $[q_i, a]$ 表示正被带头扫描的包含字符 a 的带上的位置。 $[*, a]$ 中第一个位置上的星号表示带头不扫描该字符。我们可以用 $s(n)$ 的序列来表示输入长度为 n 时机器 M 计算的任何一个机器配置。

下列字符串表示输入为 $w = a_1, \dots, a_n$ 时 M 的初始配置:

$$[q_0, B][*, a_1][*, a_2] \dots [*, a_n][*, B]^{s(n)-n-1},$$

其中的指数代表 $[*, B]$ 的 $s(n) - n - 1$ 个拷贝的连接。输入后额外的空白生成 $s(n)$ 个带方格, 这是一个计算所要求的空间上界。我们将表示每个正好有 $s(n)$ 个字符的配置。 M 的计算的表示包括用符号 \vdash 分开的机器配置序列。

现在, 我们设计一个正则表达式 α_w , 使其包含 Σ 上所有不在接收 w 的计算中出现的字符串。如果我们成功地构造出这样一个表达式,

$\alpha_w \neq \Sigma^*$ 当且仅当存在一个接收 w 的 M 的计算

当且仅当 $w \in L(M)$

当且仅当 $w \in L$

因此, 一个判定 L_{REG} 的算法能够用于判定字符串 w 是否在 L 中. α_w 的构造利用了 M 的计算的空间边界.

一个 Σ 上的表示接收 w 的计算的字符串必须满足下面三个条件:

1. 输入为 w 时, 前 $s(n)$ 个字符必须表示 M 的初始配置.
2. 符号 \vdash 分隔配置, 并且每个配置必须跟随 M 发生转换前的前驱配置.
3. 最后的配置必须是一个接收状态.

对于以上的每个条件, 我们构造一个正则表达式使其包含 Σ 上的不满足该条件的字符串. 这些表达式的并定义了一个字符串集合, 这些字符串不是输入 w 时 M 的接收计算的表示.

如果一个字符串的第一个字符不是 $[q_0, B]$, 或者它的第一个字符与 $[q_0, B]$ 匹配但是第二个字符不是 $[*, a_1]$, 或者它的前两个字符匹配但是第三个字符不是 $[*, a_2]$, 依此类推, 那么该字符串不满足第一个条件. 恰好 $s(n)$ 个上述形式的语句描述了不匹配初始配置的字符串. 该正则表达式的语言

546

$$\begin{aligned} \alpha_1 = & (\Sigma - \{[q_0, B]\}) \Sigma^* \\ & \cup [q_0, B] (\Sigma - \{[*, a_1]\}) \Sigma^* \\ & \cup [q_0, B] [*, a_1] (\Sigma - \{[*, a_2]\}) \Sigma^* \\ & \vdots \\ & \cup [q_0, B] [*, a_1] [*, a_2] \cdots [*, a_{n-1}] (\Sigma - \{[*, a_n]\}) \Sigma^* \\ & \cup [q_0, B] [*, a_1] [*, a_2] \cdots [*, a_{n-1}] [*, a_n] (\Sigma - \{[*, B]\}) \Sigma^* \\ & \vdots \\ & \cup [q_0, B] [*, a_1] [*, a_2] \cdots [*, a_{n-1}] [*, a_n] [*, B]^{s(n)-n-2} (\Sigma - \{[*, B]\}) \Sigma^* \end{aligned}$$

生成这些字符串. 符号 $(\Sigma - A)$ 是一个缩写, 它表示删除了 A 中的元素后的字母表的子集.

正则表达式

$$\alpha_2 = (\Sigma - \{[q_i, a] \mid a \in \Gamma_M, q_i \in F\})^*$$

生成了不包含接收状态的字符的所有字符串.

第二个条件要求成功的配置可以表示为 M 的一个转换. 由于每个机器配置都正好有 $s(n)$ 个符号, 所以我们可以构造一个正则表达式 α_2 , 使得 α_2 中带上 $s(n)+1$ 位置部分的符号与转换的结果不一致. 描述向右移动的转换 $\delta(q_i, a) = [b, q_j, R]$ 生成一个下列形式的子字符串:

$$\cdots [*, x] [q_i, a] [*, x] \cdots \vdash \cdots [*, x] [*, b] [q_j, x] \cdots,$$

其中 $[q_i, a]$ 和 $[*, b]$ 被 $s(n)$ 个符号分隔开.

对于每个转换 $\delta(q_i, a) = [b, q_j, R]$, 正则表达式

$$\bigcup_{x \in \Gamma_M} \Sigma^* [q_i, a] [*, x] (\Sigma^{s(n)} (\Sigma - [q_j, x]) \Sigma^* \cup \Sigma^{s(n-1)} (\Sigma - [*, b]) \Sigma^*)$$

生成的字符串与转换的结果不同. 这个表达式生成的字符串中有 $[q_i, a]$ 和 $[*, x]$, 并且在后面的 $s(n)+1$ 个位置上除了 $[*, b]$ 和 $[q_j, x]$ 以外的字符. 因此, 一个满足这个条件的字符串不是 M 的计算的表示. 类似地, 我们可以从描述了向左移动的转换中获取正则表达式. 正则表达式 α_3 是每个转换的表达式式的并.

从空间边界的标准图灵机到正则表达式的转换可以说明 L_{REG} 是 \mathcal{P} -空间完全的. 设 L 是 \mathcal{P} -空间中的语言. 那么 L 可以被多项式空间边界 $p(n)$ 的标准图灵机接收, 并且该图灵机没有始于接收状态的转换 (练习 10). 对于每个长度为 n 的字符串 w , 我们必须说明结果正则表达式的大小随 n 多项式地增长. 正则表达式 α_1 是 $p(n)$ 子表达式的并, 每一个的大小都是 $O(p(n))$. α_2 中的子表达式也是 $O(p(n))$ 的, 并且子表达式的数量与输入的长度无关. 最后, α_3 的大小是一个由 M 的状态和带字符数决定的常量. 因此 $\alpha = \alpha_1 \cup \alpha_2 \cup \alpha_3$ 的长度随输入 w 的长度多项式地增长. 上述证明说明, L_{REG} 比

547

\mathcal{P} -空间类难 综合这个结论和 L_{REG} 成员资格问题的多项式空间判定过程, 我们可以得到结论:

定理 17.5.3 语言 L_{REG} 是 \mathcal{P} -空间完全的。

由于 \mathcal{NP} 和 \mathcal{P} -空间的包含关系, 所以每个 \mathcal{P} -空间完全问题也是 \mathcal{NP} -难的。因此 L_{REG} 是一个不存在已知的非确定的多项式时间解的 \mathcal{NP} -难问题的例子。

17.6 一个难解问题

\mathcal{NP} -完全问题重要性的一个度量是在不同的问题领域和应用中遇到它们的频率。即使不存在已知的能够解决这些问题的多项式时间算法, 我们也不能得出它们不在 \mathcal{P} 内的结论。一般来说, 证明一个语言或问题在一个复杂性类中比证明它不在一个类中更容易实现。我们可以使用“猜想和检查”策略说明可满足性问题、哈密尔顿回路问题和顶点覆盖问题在 \mathcal{NP} 内时都比较简单。到目前为止, 没有人能证明任何一个这样的问题不在类 \mathcal{P} 中。

难度不同的原因在于生成一个足够说明问题在 \mathcal{P} 、 \mathcal{NP} 还是 \mathcal{P} -空间类中的算法。证明一个问题不在这些类中需要为所有解决这些问题的算法生成一个更小边界的时间或空间复杂性。本节我们将会看到识别 L_{REG} 问题的一个变种是难解的, 也就是说这是在 \mathcal{P} 外的证明。事实上, 我们会说明它在 \mathcal{P} -空间类之外, 因此不在 \mathcal{P} 或 \mathcal{NP} 中。

第2章介绍了通过给正则表达式标准定义再增加一个构造就可以得到带平方的正则表达式族。字母表 Σ 上带平方的正则表达式是通过对于每个 $a \in \Sigma$ 的 λ 、 \emptyset 和 a 递归定义的。如果 u 和 v 是 Σ 上带平方的正则表达式, 那么 $(u \cup v)$ 、 (uv) 、 (u^*) 和 (u^2) 也是 Σ 上带平方的正则表达式。像前面一样, 我们使用关联和操作符优先来减少圆括号的数量。

由于表达式 u^2 与 uu 表示同样的语言, 所以平方的增加不会增加能够用正则表达式表示的语言。但是, 平方操作符减少了描述一个语言的表达式的长度。平方操作符允许我们用 $O(n)$ 个字符把一个正则表达式的 2^n 个拷贝的连接写做

$$(\dots((u^2)^2)\dots)^2,$$

其中使用了 n 次平方操作符。由于复杂性把输入和时间及空间关联起来了, 所以复杂性度量的增加会导致一个更加紧凑的输入表示。

我们将说明判定一个带平方的正则表达式的语言是否不包含字母表上所有的字符串问题不在 \mathcal{P} -空间内。这与前一节考虑的问题相同, 而唯一的不同是平方操作符出现在正则表达式中。后续部分介绍了该问题的证明, 该证明中把图灵机计算表示成了带平方的正则表达式。

设 L 是一个能被空间边界为 2^n 的图灵机接收的语言, 且 L 不能被空间边界为 2^{n-1} 的图灵机接收。定理 17.5.3 保证了存在这样一种语言。设 M 是一个接收 L 且空间复杂性 $sc_M(n) = 2^n$ 的单带确定型图灵机。像前面一样, M 的计算可以表示成字母表 $\Sigma = \{ (q_i, a), [*, a_i], \vdash \mid q_i \in Q, a \in \Gamma \}$ 上的正则表达式。对于每个 Σ^* 中的 $w = a_1 \dots a_n$, 我们定义一个正则表达式 α_w , 使其语言是不在 M 接收 w 的计算中出现的字符串。 α_w 构造使用的方法同 17.5 节, 但是我们使用平方来保证 α_w 的长度随 w 的长度线性地增长。

在 17.5 节中, 每一个在 α_w 中编码的机器配置都有 $s(n)$ 个带方格, 其中 $s(n)$ 是机器 M 的空间边界。为了数字操作的方便, 我们在这里选择带位置 $2^n + n + 1$ 。输入为 w 时, 表示 M 的计算的初始配置的字符串包括:

$$[q_0, B][*, a_1][*, a_2] \dots [*, a_n]$$

随后是 $[*, B]$ 的 2^n 个拷贝。平方操作符使我们能够用一个长度为 $O(n)$ 的正则表达式描述这个字符串。通过检查, 我们可以看到当我们用平方来表示 $[*, B]$ 和 Σ 的指数的重复时, 子串 α_1 、 α_2 和 α_3 只需要大小为 $O(n)$ 的空间。因此, α_w 的长度是 $O(n)$ 。

设 L_{REG2} 表示正则表达式集合, 其元素使得 $\alpha_w \neq \Sigma^*$ 并且是 α_w 的带平方的形式。

定理 17.6.1 语言 L_{REG2} 是难解的。

证明: 假设 L_{REG2} 的成员资格问题可以由一个多项式空间边界的图灵机 M' 判定。结合 α_w 的构造和

M' 的计算可生成如下操作序列:

1. 输入: 一个长度为 n 的字符串 $w \in \Sigma_M^*$
2. 转移: 正则表达式 α_w 的构造
3. M' 的计算: 判定是否 $\alpha_w \neq \Sigma^*$
4. 结果: $w \in L$ 当且仅当 M' 接收 α_w 。

整个过程可以在多项式空间内完成并且接收语言 $L(M)$ 。由于 $L(M)$ 不能被任何空间复杂性小于 $2^{n/2}$ 的图灵机接收, 所以这是一个矛盾。

语言 L_{REG} 显然是可判定的。一个简单的策略是扩展 α_w 中平方的出现来为同一个语言生成一个标准正则表达式。根据练习 13, 结果表达式的问题可以在长度的多项式空间内得到回答。不幸的是, 后一种表达式的空间可能随着 w 的长度指数级地增长。

17.7 练习

1. 设 Q 是可以在多项式时间内归约到语言 L 的语言。请证明 Q 在多项式时间内可以归约到 \bar{L} 。
2. 设计一个空间复杂性为 $Q(\log_2(n))$ 的双带图灵机, 使之能够接收 $\{a^i b^i \mid i \geq 0\}$ 。
3. 设 L 是一个可以被图灵机 M 接收的语言, 且输入长度为 n 时 M 的计算最多需要大小为 $s(n)$ 的空间。注意我们不要求 M 的所有计算都能终止。请证明 L 是递归的。
4. 说明 \mathcal{P} -空间对于并和补是封闭的。
5. 给下列每个空间边界设计一个图灵机, 说明该函数是完全空间可构造的。
 - a) $s(n) = n$
 - b) $s(n) = 3n$
 - c) $s(n) = n^2$
 - d) $s(n) = 2^n$
6. 设 M 是空间复杂性 $sc_M(n) = f(n) \geq n$ 的图灵机。请注意这表示存在某个长度为 n 的输入使得 M 正好使用 $sc_M(n)$ 个带方格。说明 $f(n)$ 是完全空间可构造的。
7. 设计一个输入字母表为 $\{1\}$ 的单带确定型图灵机, 使其在处理长度 $n > 1$ 的输入时正好使用 2^n 个带方格。
8. 设 $s(n)$ 是完全空间可构造的函数, $s(n) \geq n$ 且 $s(0) > 0$ 。说明存在对任意长度为 n 的输入都正好使用 $s(n)$ 个带方格的单带图灵机。
9. 设 M 是一个 $s(n)$ 边界的图灵机且 $s(n) \geq n$ 。证明存在一个接收 $L(M)$ 的 $s(n)$ 边界的单带图灵机。
10. 设 L 是一个 \mathcal{P} -空间中的语言。证明存在一个没有始于接收状态的转移的单带图灵机, 该图灵机接收 L 并且其计算有多项式空间边界。
11. 请证明能被 $s(n) = \log_2(n)$ 空间边界图灵机接收的语言集合是能被 $s(n) = n$ 空间边界图灵机接收的语言集合的子集。
12. 能被 $s(n) = n'$ 空间边界图灵机接收的语言集合是否是 $s(n) = 2n'$ 空间边界图灵机接收的语言集合的子集? 请证明你的结论。
13. 说明语言 L_{REG} 在 \mathcal{P} -空间内。提示: 使用 \mathcal{P} -空间和 NP -空间的等价, 设计一个能够判定 L_{REG} 成员资格问题的非确定型多项式空间边界的图灵机。
14. 请证明定理 17.5.2。
15. 说明任意 \mathcal{P} -空间完全的语言都是 NP -难的。

参考文献注释

Ladner [1975] 证明了假设 $\mathcal{P} \neq NP$ 成立, NP 语言的存在性。Karp [1972] 给出了 \mathcal{P} -空间完全性的第一个证明。本书对空间复杂性介绍是按照 Hopcroft 和 Ullman [1979] 中的内容给出的。这本著作和 Papadimitriou [1994] 给出了很多关于空间复杂性的其他结果。判定扩展正则表达式语言的难解性来自 Meyer 和 Stockmeyer [1973] 的著作。

549

550

551

552

第五部分

确定型语法分析

程序设计语言的定义和程序编译将形式语言理论和计算机科学应用程序直接联系起来。程序的编译是一个多步骤的过程，在编译的过程中，使用高级语言书写的源代码经过分析和转换变成可执行的机器语言代码或者汇编语言代码。这一过程的头两个步骤是词法分析（lexical analysis）和语法分析（parsing），这两个步骤主要用于检查源代码语法的正确性。词法分析器阅读源代码中的字符，并构造出程序设计语言的符号（token）（保留字，标识符，特殊符号等）的序列。然后，语法分析器（parser）判断作为结果的符号串是否能够满足程序设计语言定义的语法要求。

1960年，ALGOL 60诞生，这是第一种利用文法规则来对语法进行形式定义的程序设计语言。从此，文法成为定义程序设计语言语法的主要工具。附录Ⅲ中给出了Java程序设计语言的巴克斯-诺尔范式（Backus-Naur form）文法，该文法用于定义一个在语法上正确的Java程序构成的集合。但是，如何判断某个Java语言的源代码序列是否构成了一个语法正确的程序呢？如果源代码是可以通过使用文法规则从变量（variable）（编译单位）推导出来，那么相应的语法就是正确的。要判断使用Java语言或者其他使用上下文无关文法（context-free grammar）定义的语言所书写的程序在语法上是否正确，就必须设计语法分析算法，该算法可以生成一种文法语言的推导串。如果该字符串不在相应的语言中，那么这些程序就会发现没有相应的推导存在。

第18章将会演示用相应算法进行语法检查（syntax checking）的可行性。同时，这一章将会引入自顶向下语法分析（top-down parsing）和自底向上语法分析（bottom-up parsing）两种方法，这两种方法均是通过搜索可能存在的推导图（graph of possible derivation）来进行的。分析器进行穷举搜索（exhaustive search）；自顶向下分析器检查所有可允许的规则应用，而自底向上分析器则进行所有可能的归约（reduction）。无论哪种情况，这些算法都能够很好地检测很多无关系的推导。虽然这些算法能够演示其语法分析的可行性，但它们的效率低下问题使得它们对于商用编译器或者解释器来说都是不可接受的。

19和20章引入了两类上下文无关文法，这两类文法使得语法分析变得更加有效。为了确保所选择的动作是合适的，分析器“预读（look ahead）”将要被分析的字符串。如果在每一个步骤中都有至多一个规则可以成功地扩展当前的推导，那么这个时候的分析器就是确定型的。LL(k)文法允许确定型的自顶向下分析器预读 k 个符号。在自底向上分析器中，LR(k)分析器使用有限自动机，并且预读 k 个符号来选择归约或者移进（shift）。当代大多数程序设计语言都是使用LL文法或者LR文法定义的，也有使用这两种文法的变种来定义的，从而使得分析更加有效。贯穿整个语法分析的介绍中，我们都假定文法是无二义性的，这对于用来定义任何程序设计语言的文法来说都是合理的。

[553]

[554]

第 18 章 语法分析引论

本章将引入两个简单的分析算法,用以显示自顶向下分析和自底向上分析的特性。这些算法均是基于图的宽度优先搜索(breadth-first search)的,图中的路径代表了文法的推导。分析器的输入是文法字母表上的一个串,如果该串属于这个文法所定义的语言,那么预期结果则是输出该输入串的一个推导。否则,分析器将会显示无法推导出该输入串。

自顶向下分析起始于文法的开始符号(start symbol),并系统地应用相应的规则以尝试产生输入串。自底向上分析将该过程颠倒过来;它起始于输入串自身,并且“逆向”应用规则,以尝试能够产生开始符号。这些简单的算法演示了用于语法分析的规则形式的潜在有效性。对于任何一种语言来说,搜索可能不会终止。然而,使用格立巴赫范式的文法可以停止自顶向下的算法,而不具有链规则的非限定文法则足以保证能够停止一个自底向上的分析器。

用于定义程序设计语言的文法在规则的问题上,要求一些附加的条件以便有效地分析语言串。为了有效分析而特别设计的文法将会在第 19 章和第 20 章中介绍。

18.1 文法图

{555}

本章的引论很直观,自顶向下分析被描述成搜索一个推导图。因为任何一个可推导的终结字符串都具有一个最左推导(见定理 3.5.1),所以这里将会把搜索限制在最左推导。如果文法是无二义性的,那么这种推导就会生成一棵树,这棵树的树根就是文法的开始符号。这里需要指出一点,对于任何一种有意义的文法来说,都将会无限多种推导,而且图中会有无限多个节点。

定义 18.1.1 设 $G=(V, \Sigma, P, S)$ 是上下文无关文法,文法 G 的图 $g(G)$ 是一个带标记的有向图,其中节点和弧的定义如下:

- i) $N = \{w \in (V \cup \Sigma)^* \mid S \xRightarrow{*} w\}$
- ii) $A = \{[v, w, k] \in N \times N \times N \mid v \xRightarrow{k} w \text{ 应用规则 } k\}^{\odot}$ 。

图中的节点是文法的左句型,也就是可以通过最左推导从开始符号推导出来的串。在 $g(G)$ 中,如果 $v \xRightarrow{k} w$,也就是说如果 w 可以利用一个最左规则应用从 v 处获得,则串 w 邻接于 v 。这些文法规则被依次标上号码,这些号码用于标记图中的弧线以及后来的语法分析算法。如果用规则 k 的应用来创建从 v 到 w 的弧线,那么这个弧线就被 k 所标记。在 $g(G)$ 中,从 S 到 w 的一条路径代表了从 S 到 w 的一个最左推导。

文法图是为任何一种上下文无关文法定义的。如果文法是无二义性的,那么结果图就是一棵将开始符号作为根的树。因为可以进行确定型语法分析的文法是无二义性的,所以当描述语法分析策略的时候,就可以任意使用树的术语表和树的搜索算法。这里特别需要指出的是, $g(G)$ 被称为文法 G 的推导树。

在 $g(G)$ 中,推导用路径来表示,那么,判断串 w 是否在语言 G 中这个问题就简单地转化为在 $g(G)$ 中寻找一条从 S 到 w 的路径。图 18-1 阐明了如何在图中将推导表示为路径,该图使用了文法 AE (加法表达式):

○ $N \times N \times N$ 中的前两个 N 是 (i) 中定义的节点集合 N ; 最后一个 N 是自然数集合。——译者注

1. $S \rightarrow A$
2. $A \rightarrow T \mid A + T$
3. $A \rightarrow A * T$
4. $T \rightarrow b$
5. $T \rightarrow (A)$.

AE 的开始符号是 S ，这个语言包含了所有的由运算符 $+$ 、 $*$ 、 $()$ 、 b 以及圆括号构成的算数表达式。由 AE 生成的串包括 b 、 $((b))$ 、 $(b+b)$ 以及 $(b) + b$ 等等。本章将会使用文法 AE 来阐明语法分析算法的特性。

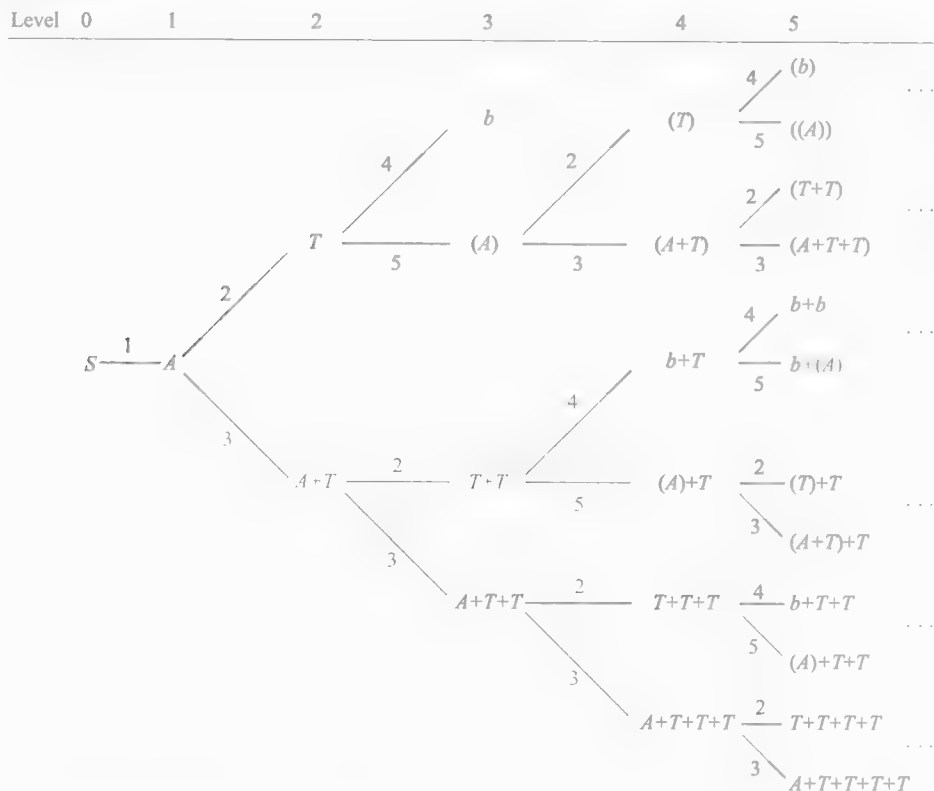


图 18-1 AE 的推导树

句型中最左边的变量相应的规则（即以该变元为左部的规则）个数决定了该节点的子节点个数。任何一个直接递归或者间接递归的存在都会在树中产生无穷多的节点。直接递归规则 A 和间接递归规则 T 的反复应用能产生图 18-1 中的树的任意长度的路径。

标准的树搜索技术可以用于检查推导树中的推导。在关于树搜索的术语中，该推导树被称为隐含树（implicit tree），这是因为这种树的节点并不是搜索前构造起来的。这种搜索在检查路径时构造出推导树。这种算法的一个重要特征就是要显式地构造尽可能少的隐含树。

18.2 自顶向下语法分析

文法推导树中的路径代表了文法的最左推导。这里的自顶向下语法分析算法使用了宽度优先策略来搜索隐含树中的输出串的推导。如果发现了串的一个推导，则算法接收该输入；如果语法分析器确定没有相应的推导，则算法拒绝该输入。

为了限制所需要的搜索数量，语法分析器使用前缀匹配来确定不能出现在输出串推导中的句型串的终结前缀（terminal prefix）是出现在最左变量前边的子串。也就是说，如果 B 是串中第一个变

[557] 量, 那么 x 就是 xBy 的终结前缀。若串 xBy 的终结前缀 x 不能与输入串的前缀相匹配, 则输入串不能从 xBy 推导出来。这样的串 xBy 被成为死端 (dead end), 并且搜索时要忽略其后代。

语法分析器构造出一棵搜索树 T , 指针从子节点指向其父节点 (父指针)。搜索树是隐含树的一部分, 隐含树在语法分析的过程中需要进行明确的检查。用号码对文法规则进行标记, 节点的子节点则会根据规则的排序而被加入到树中。生成一个节点的后继的过程, 以及把它们加入到搜索树中则被称为扩展节点 (expanding the node)。

宽度优先树搜索需要使用先进先出的内存管理策略, 这里使用队列来实现这一策略。队列 Q 的维护需要提供三个函数: $INSERT(x, Q)$ 表示将串 x 放到队列的尾部; $REMOVE(Q)$ 返回队首的元素, 并将其从队列中删除; $EMPTY(Q)$ 是一个布尔函数, 如果队列为空, 则该函数返回 true; 否则, 返回 false。

算法 18.2.1 宽度优先自顶向下语法分析器

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

数据结构: 队列 Q

1. 用根 S 初始化 T

$INSERT(S, Q)$

2. repeat

2.1. $q := REMOVE(Q)$ (要扩展的节点)

2.2. $i := 0$ (最后用到的规则的序号)

2.3. $done := false$ (扩展完成的布尔型标记)

设 $q = uAv$, 其中 A 是 q 中最左边的变量

2.4. repeat

2.4.1. if 没有规则序号比 i 大的规则 A , then $done := true$

2.4.2. if not done then

设 $A \rightarrow w$ 是第一个序号比 i 大的规则 A , 且设 j 是该规则的序号

2.4.2.1. if $uwv \notin \Sigma^*$ 并且 uwv 的最终前缀与 P 的一个前缀匹配

2.4.2.1.1. $INSERT(uwv, Q)$

2.4.2.1.2. 在 T 中添加节点 uwv . 设置一个从 uwv 到 q 的指针.

end if

end if

2.4.3. $i := j$

until done or $p = uwv$

until $EMPTY(Q)$ or $p = uwv$

3. if $p = uwv$ then 接收 else 拒绝

[558]

对搜索树进行初始化是从树根 S 开始的, 这是因为自顶向下的算法尝试找到从 S 开始的输入串 p 的推导。这个算法包括两个嵌套的 repeat-until 循环。外部循环选择队列中第一个节点 q 以便进行扩展。在步骤 2.4 中, 内部循环按照规则的编号方式所指定的序列生成节点 q 的后继。串 uAv 扩展生成的串 uwv 存在三种可能性: 它可能是一个终结串, 可能是一个死端, 或者可能是一个需要进一步扩展的句型。

如果 uwv 是一个终结串, 那么就表示该推导已经结束了, 并且该串不会被加入到树中, 也不会被加入到队列里。语句 until 则检查该串是否是输入串。如果是的话, 就停止计算并且接收该串。否则, 继续生成下一个子节点以对 uAw 进行扩展。

步骤 2.4.2.1 负责检查前缀匹配。如果某个串是死端，那么该串将不会被加入到队列或者树中，在步骤 2.4.2.1.1 和步骤 2.4.2.1.1 中，满足前缀匹配的串被加入到队列和树中。无论上述哪种情况，扩展都是通过生成 uAv 的下一个子节点来进行的。

节点选择的周期和扩展的过程是反复进行的，除非生成了输入串，或者队列为空，否则这个过程将一直持续下去。只有当所有可能的推导均被检查过，并且没有成功推导出输入串的时候，后面那种情况才会出现。队列所维护的先进先出顺序生成了搜索树的宽度优先结构。

图 18-1 给出了文法 AE 的推导树的前五层。分析器逐层评估树中的这些节点。图 18-2 给出了分析 $(b+b)$ 所构造的搜索树。这里用点线来表示那些已经生成，但是未被加入到搜索树中的句型。

[559]

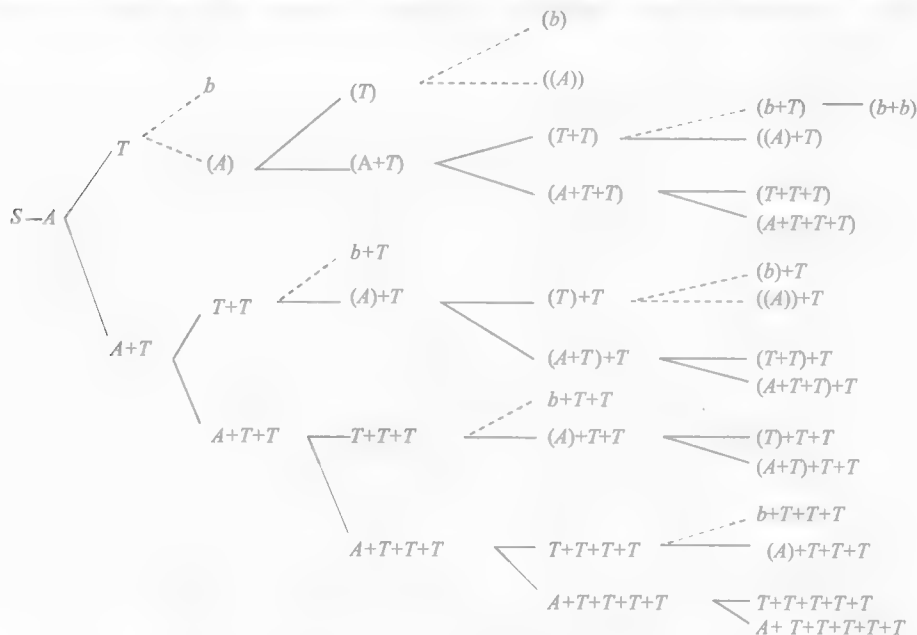


图 18-2 $(b+b)$ 的自顶向下语法分析。

步骤 2.4.2.1 进行了比较，该比较用于检查语法分析器生成的句型的终结符前缀是否与输入串相匹配。为了获取匹配所需要的信息，语法分析器在其构造推导的过程中“读取”输入串。语法分析器按从左到右的方式扫描输入串，一直到导出句型的最左变量。终结前缀的增长使得语法分析器可以读取整个输入串。语法分析器扫描的串的初始段和导出串的终结符前缀之间存在相应的关系， $(b+b)$ 的推导可以展现出这一点：

推导	分析器读入的输入
$S \Rightarrow A$	λ
$\Rightarrow T$	λ
$\Rightarrow (A)$	$($
$\Rightarrow (A+T)$	$($
$\Rightarrow (T+T)$	$($
$\Rightarrow (b+T)$	$(b+$
$\Rightarrow (b+b)$	$(b+b)$

语法分析器不仅要能生成语言串的推导，还要具备判定该串不在此语言中的能力。图 18-2 中的搜索树的下部可以潜在地无限增长。规则 $A \Rightarrow A+T$ 的直接递归可以构造出任何一个或者多个以 $+T$ 为后缀的串。当搜索一个不在该语言中的串的推导时，直接递归规则 A 永远不会生成可以终止此搜索的前缀。

读者可能存在这样的疑问，串 $A+T+T$ 是不能推导出 $(b+b)$ 的，并且该串不能声明一个死端。两个 $+$ 的存在的确能够确保没有规则序列可以将 $A+T+T$ 转换成 $(b+b)$ 。然而，做出这样的判断需要对已被语法分析器扫描过的初始段之外的输入串有所了解。第 19 章中的语法分析器将会“预读”串，越过语法分析器生成的终结前缀进行扫描，以便于帮助选择语法分析器将要进行的后续动作。

对于进入永不停止的计算这种情况来说，它是由一种特殊的规则的存在所引起的，该规则的应用不会增加终结符前缀的长度。一种“修改”算法 18.2.1 的方法是只使用不允许这种情况发生的文法。

560

在第4章中,我们可以看到,任何一种上下文无关的语言都是由乔姆斯基范式中的文法生成的。乔姆斯基范式文法中的每个规则的应用,均是完成下面这两件事情中的一种:要么将终结符加到导出串的前缀上,要么完成一个推导。这就保证了算法 18.2.1 对于任何输入串都会停止,这是因为外在搜索树将会拥有一个深度,而这个深度受到输入串的长度的限制。

对于为语言中任何一个串构造推导来说,使用宽度优先算法都将是会成功的,尽管如此,这种方法的实际应用存在很多缺点。冗长的推导以及带有大量规则的文法导致了搜索树的大小增长迅速。搜索树的指数速度的增长并未受到语法分析算法的限制,但却是宽度优先树搜索的一个普遍特性。如果将文法设计成可以利用前缀快速匹配的情况,或者可以改进其他的情况以找到搜索中的死端,那么这种与搜索树增长问题相关联的组合问题可能就会得到延缓,但是这种问题是不可能避免的。所以,我们需要更好的策略来解决这一问题。

18.3 归约和自底向上语法分析

在自顶向下分析中,寻找一个推导需要从开始符号检测文法推导树中的路径。这种搜索可以系统地构造推导,直到找到输入字符,或者,如果发现无相应的推导存在,则可以减少此输入。这种策略执行的是一种穷举搜索。除了因为死端的定义而导致的修剪,每个输入串均会生成相同的树。用这种方式来搜索需要检查出很多不可能生成输出串的推导。例如,图 18-2 中,以 $A+T$ 为根的整个子树包含了不能产生 $(b+b)$ 的推导。

自底向上分析会构造出一棵根部是输入串 p 的搜索树,并且可以“逆向”使用规则。当从输入串开始进行搜索的时候,只是对能够生成 p 的推导进行检查。这就使得注意力集中在搜索上,并且这样能够减少搜索树的大小。为了限制隐含图的大小,自顶向下语法分析器只是生成了最左推导。因为自底向上语法分析器逆向构造推导,所以只要检查其最右推导即可。自底向上语法分析可以被看成是对于包含所有从 p 开始进行最右推导的串的隐含图的搜索。

	规则应用	归约
字符串	uAv	uvw
规则	$A \rightarrow w$	$A \rightarrow w$
结果	uvw	uAv

与构造推导相反的操作,这里称之为归约。正如所预料的那样,规则应用和归约具有相反关系:归约利用左边的单个变量来代替规则的右部。就像归约这个名字所暗示的那样,归约的目的就是减少字符串的长度,这一点在下面的例子中会加以阐明。

561

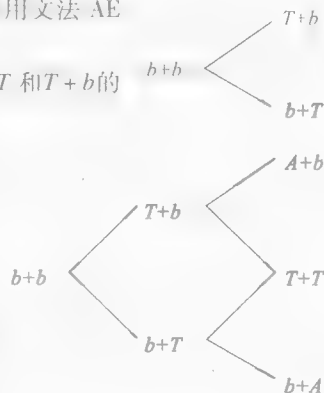
字符串	abb	$aAbAbbab$	BAA
规则	$A \rightarrow ab$	$A \rightarrow bAb$	$A \rightarrow AA$
归约	Ab	$aAAbab$	BA

每当规则右部的长度大于一的时候,归约就生成一个长度更短的串。

在某些情况下,必须要能够确保搜索只是检查最右推导,这里使用文法 AE 来阐述这种情况。考虑使用规则 $T \rightarrow b$ 对串 $b+b$ 进行归约的情况:这棵树显示了推导 $T+b \Rightarrow b+b$ 和 $b+T \Rightarrow b+b$ 的情况。通过增加 $b+T$ 和 $T+b$ 的所有归约来构造另外一层,从而生成了下面这棵树:

这里需要注意的是,串 $T+T$ 出现了两次,一次是在推导 $T+T \Rightarrow T+b \Rightarrow b+b$ 中出现,另外一次是在 $T+T \Rightarrow b+T \Rightarrow b+b$ 中出现。后面的这个推导并不是最右推导,搜索不应该考虑相应的归约。

只有当串 v 没有变量的时候,规则 $A \rightarrow w$ 对 uvw 产生的归约才会生成一个最右推导。如果 v 中存在一个变量,则相应的推导 $uAv \Rightarrow uvw$ 就不是最右推导,这是因为 v 中的变量发生在 A 的右边。这种情况合并并在自底向上的语法分析器中,从而可以保证最右推导的生成。例 18.3.1 就描述了从归约序列中获得最右推导的过程。



例 18.3.1 串 $(b) + b$ 归约到开始符号 S 是通过使用文法 AE 的规则给出的。

使用规则 $T \rightarrow (A)$ 和 $A \rightarrow A + T$ 的归约缩短了串的长度, 并且 $T \rightarrow b$ 将终结符 b 转换成变量 T 。在将 w 归纳到 S 的时候, 颠倒句型的序列会产生最右推导。

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow A + T \\ &\Rightarrow A + b \\ &\Rightarrow T + b \\ &\Rightarrow (A) + b \\ &\Rightarrow (T) + b \\ &\Rightarrow (b) + b. \end{aligned}$$

归约	规则
$(b) + b$	
$(T) + b$	$T \rightarrow b$
$(A) + b$	$A \rightarrow T$
$T + b$	$T \rightarrow (A)$
$A + b$	$A \rightarrow T$
$A + T$	$T \rightarrow b$
A	$A \rightarrow A + T$
S	$S \rightarrow A$

因为推导的构造在遇到开始符号的时候就结束了, 所以自底向上语法分析器经常逆向构造最右推导。

18.4 自底向上语法分析器

通过自底向上语法分析器搜索的隐含图是由文法 $G = (V, \Sigma, P, S)$ 和输入串 p 一同决定的。图中的节点表示的是一些串, 这些串可以通过应用最右规则推导出来。如果节点 w 可以通过应用某个最右规则而从 v 处获得, 那么节点 w 与节点 v 邻接。

宽度优先自底向上语法分析器以 p 为根逐层构造出一棵搜索树。和自顶向下语法分析器一样, 搜索树 T 是使用队列操作 *INSERT*, *REMOVE* 和 *EMPTY* 构造起来的。

算法 18.4.1

宽度优先自底向上语法分析

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

数据结构: 队列 Q

1. 用根 P 初始化 T

INSERT (p , Q)

2. repeat

$q := \text{REMOVE}(Q)$

2.1. for P 中的每个规则 $A \rightarrow w$ do

2.1.1. for 对于每个 $v \in \Sigma^*$ 的 P 的分解 uvw do

2.1.1.1. *INSERT* (uAv , Q)

2.1.1.2. 在 T 中增加结点 uAv . 设置一个从 uAv 指向 q 的指针.

end for

end for

until $q = S$ or *EMPTY*(Q)

3. if $q = S$ then 接收 else 拒绝

搜索树从树根 p 开始进行初始化。算法的余下部分包括以下这些过程: 选择节点 q 从而进行扩展、生成 q 的归约, 以及更新队列和树。步骤 2.1.1 用于检查已经被归约的串 w 的右边没有变量, 从而保证只有最右推导被插入到队列中, 并且该推导被加入到搜索树里。

图 18-3 显示了当串 $(b + b)$ 使用自底向上语法分析器来进行语法分析的时候所构造出的搜索树沿着从 S 到 $(b + b)$ 的路径生成了最右推导树, 如下所示:

[562]

[563]

$$\begin{aligned}
 S &\Rightarrow A \\
 &\Rightarrow T \\
 &\Rightarrow (A) \\
 &\Rightarrow (A + T) \\
 &\Rightarrow (A + b) \\
 &\Rightarrow (T + b) \\
 &\Rightarrow (b + b).
 \end{aligned}$$

图 18-3 中, $(b + b)$ 使用自底向上语法分析法生成搜索树; 图 18-2 中, $(b + b)$ 使用自顶向下语法分析法生成搜索树, 将这两者相对比, 将搜索限制到可以产生 $(b + b)$ 的推导, 这样可以极大地减少产生的节点的数目。

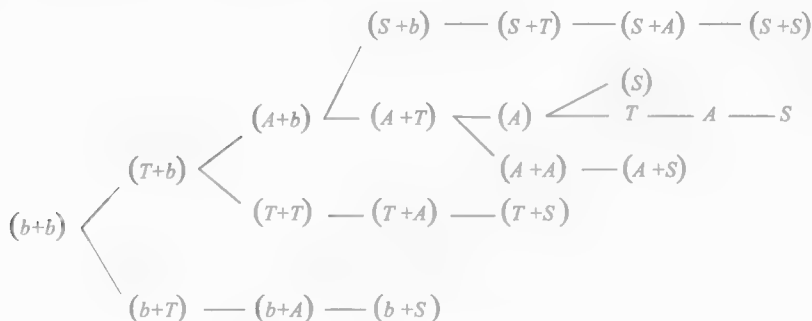


图 18-3 $(b + b)$ 的自底向上语法分析

使用自顶向下和自底向上的方法所生成的搜索树的大小是不同的, 这些关于搜索树大小的不同点生动地体现于不在语言中的那些串。图 18-4 显示了对于串 $(b +)$ 的语法分析而产生的树。根据 AE 中的最左推导, 自顶向下的分析永远不会终止。检查四个节点之后, 自底向上的分析就会暂停。

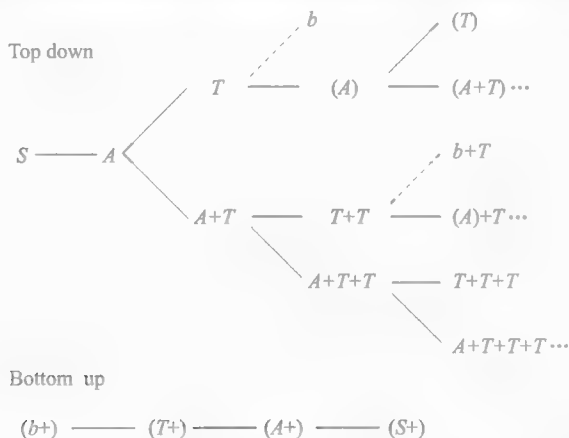


图 18-4 $(b +)$ 的自顶向下语法分析和自底向上语法分析

自顶向下语法分析永远不会停止, 自底向上语法分析在检查 4 个节点之后停止。

前面的表述省略了一个非常重要的步骤——寻找串 q 的归约。现在将要调整这个漏洞。串 q 已经具备了一个归约, 如果以下条件成立:

- i) q 可以被写为 uvw , 并且
- ii) 文法中存在一个规则 $A \rightarrow w$

确定串 q 的归约要求规则的右部需要有子串 q 。这里可以使用一种移进-比较策略, 以产生串 q 的所有归约。串 q 被分成两个子串, $q = xy$ 。初始化的划分将 x 设为空串, 并将 y 设为 q 。每条规则的

右部均与 x 的后缀进行比较。当 x 可以写成 uw 的形式并且 $A \rightarrow w$ 是文法的规则的时候, 就表示出现了匹配。这表示将 q 归约为 uAy 。

对于一个给定的对 xy , 如果所有的规则都与 x 的后缀比较过, 那么 q 就被重新划分成两个子串 $x'y'$, 并且这个过程会重复进行。这个分解将 x' 设置成与 y 第一个元素连接的 x ; y' 就是将 v 移去其首元素后余下的部分。更新分割的过程被看成是移进。在文法 AE 中, 移进和比较操作用于生成串 $(A+T)$ 的所有可能的归约。

	x	y	后缀	规则	归约
	λ	$(A+T)$	λ		
Shift	$($	$A+T)$	$(, \lambda$		
Shift	$(A$	$+T)$	$(A, A, \lambda$	$S \rightarrow A$	$(S+T)$
Shift	$(A+$	$T)$	$(A+, A+, +, \lambda$		
Shift	$(A+T$	$)$	$(A+T, A+T, +T, T, \lambda$	$A \rightarrow A+T$	(A)
				$A \rightarrow T$	$(A+A)$
Shift	$(A+T)$	λ	$(A+T), (A+T), +T), T),), \lambda$		

在生成串的归约的过程中, 规则的右部必须与 x 的后缀相匹配。所有其他发生在 x 的规则右部的归约均会在最近一次移进之前被发现。

如同前面的表中所示, λ -规则将会在每次分解 xy 的过程中与一个后缀相匹配, 并且会为任何一个长度为 n 的串产生 $n+1$ 个归约。因此, 自底向上语法分析算法不应该被用于使用 λ -规则的文法。然而, λ -规则不会产生第 20 章中所考虑的自底向上语法分析器所产生的问题。

宽度优先自底向上语法分析器对于每一个可能的输入串都会停止吗, 或者可以这样说, 该算法在 repeat-until 循环中会产生不确定的继续吗? 如果串 p 在该文法的语言中, 那么将会找到一个最右推导。如果每条规则的右部的长度均大于 1, 那么一个句型的归约将会产生一个长度小得多的新串。对于可以满足这种情况的文法来说, 搜索树的长度不能超过输入串的长度, 要么找到一个推导, 要么失败, 只有这两种情况发生, 从而可以确保语法分析的终止。然而, 这种情况并未被具有 $A \rightarrow B$ 、 $A \rightarrow a$ 和 $A \rightarrow \lambda$ 形式的规则所满足。在练习 11 中, 将会要求读者给出一个文法和串, 使得算法 18.4.1 永远不会停止。对于不存在 λ -规则和链规则的文法来说, 终止是可以保证的。

在算法 18.4.1 中, 对于某个句型来说, 生成它可能存在多种动作, 那么自底向上语法分析器的有效性将会因为这个原因而受到严重影响。例如, 在都使用规则 AE 的情况下, 串 $A+T$ 具有两个归约, 而 $b+b+b$ 具有三个归约。穷举搜索策略将会执行每一个归约, 将产生的句型加入到搜索树中去, 并且生成他们的后代。为了生成更多的分析, 就需要有能力在每一个步骤中选择一个单独的动作。允许确定型自底向上语法分析的文法将在第 20 章中介绍。

18.5 语法分析和编译

语法分析是一个检验程序源代码是否满足程序设计语言语法规则的过程。将用高级语言书写的源代码转换成为可执行的机器代码或者汇编语言代码的整个过程就是编译这个过程。编译程序包括词法分析、词法扫描, 以及代码的生成。除了语法分析, 编译过程最初的两个步骤包括语义分析以及错误鉴定和回复。这里将简单地讨论一下程序中除了包含在词法分析和词法扫描之中的语法分析之外的部分。

词法分析扫描源代码并且生成程序设计语言的记号串。程序设计语言的记号包括语言中使用的标识符、保留字、文字以及特殊符号。记号串中不包括空格键、注释、回车键、换行符以及其他源代码中的非语言成分。字符序列有可能不能按照语法形成正确的标识符、常量或者特殊符号, 词法分析也会检测出这样的错误。Java 定义的标识符要求第一个符号必须是字母、下划线或者是美元符号。当词法分析遇到一个没有满足要求的符号串, 或者与其他任何一个 Java 保留字相匹配的单词或者符号的时

候,它就会产生一条错误信息。因为程序设计语言的记号形成了规则语言,所以词法分析经常通过有限自动机来实现。

语法分析器检测词法分析器产生的记号串是否定义了一个语法正确的程序。这是通过构造一个推导来实现的,推导可以使用自顶向下和自底向上这两种方式中的任何一种,这需要看该串使用哪种文法规则来定义程序设计语言。成功的语法分析会生成程序的推导或者分析树(见3.1节)。

前两节中所展示的语法分析结果只是关于输入串的正确与否的指示——接收或者拒绝。编译器的语法分析阶段必须要定义语法错误,为文法生成尽可能多的错误信息,并且能够从错误中恢复,以继续其语法分析。声明终结符号、隔离符号以及特殊符号对于错误恢复来说毫无意义。如果在分析一个语句序列的时候检测出一个错误,那么就要生成一条错误信息,并且语法分析将会继续进行下去,直到它遇到像分号或者括弧这样表示句子终结的符号才会停止读取记号。此时,语法分析器将会尝试继续分析正在读取的记号串的剩余部分。

在语法分析的过程中,通过检测由语法分析器产生的陈述语句的语义正确性可以获得一些信息,语义分析所使用的正是这些信息。在编译这个阶段,可以被定义的语义错误包括将保留字声明为标识符,引用未被定义的变量,标识符的多次声明,以及指派或者操作过程中的类型不兼容等。

在成功地进行了词法分析和语义分析之后,语法分析树就被频繁地用于生成程序的中间语言的表述形式。设计中间表述是为了方便编译的最后一个步骤:机器语言或者汇编语言代码的优化转换。

18.6 练习

1. 构造下述文法 G 的图的子图,该文法包括少于三步的推导生成的左句型。

$$\begin{aligned} G: S &\rightarrow aS \mid AB \mid B \\ A &\rightarrow abA \mid ab \\ B &\rightarrow BB \mid ba \end{aligned}$$

2. 构造下述文法 G 的图的子图,该文法包括不多于四步的推导生成的左句型。

$$\begin{aligned} G: S &\rightarrow aSA \mid aB \\ A &\rightarrow bA \mid \lambda \\ B &\rightarrow cB \mid c \end{aligned}$$

G 是二义性的吗?

从练习3到练习7,使用文法 AE 来分析输入串,并追踪算法的行为。如果该输入串在语言中,那么给出语法分析所构造出的推导。

3. 将算法 18.2.1 中的输入变成 $(b) + b$ 。
4. 将算法 18.2.1 中的输入变成 $b + (b)$ 。
5. 将算法 18.2.1 中的输入变成 $((b))$ 。
6. 将算法 18.4.1 中的输入变成 $(b) + b$ 。
7. 将算法 18.4.1 中的输入变成 $(b))$ 。
8. 对串 $b) + b$ 进行分析,请给出由算法 18.2.1 和算法 18.4.1 生成的搜索树的前五层。
9. 设文法 G 为

1. $S \rightarrow aS$
2. $S \rightarrow AB$
3. $A \rightarrow bAa$
4. $A \rightarrow a$
5. $B \rightarrow bB$
6. $B \rightarrow b$

- a) 给出 $L(G)$ 的一个集论 (set-theoretic) 定义
- b) 给出 $baab$ 的自顶向下语法分析所构造的树
- c) 给出 $baab$ 的自底向上语法分析所构造的树

10. 设文法 G 为

1. $S \rightarrow A$
2. $S \rightarrow AB$
3. $A \rightarrow abA$
4. $A \rightarrow b$
5. $B \rightarrow baB$
6. $B \rightarrow a$

- a) 给出 $L(G)$ 的一个正则表达式。
 - b) 给出 $abbbbaa$ 的自顶向下语法分析所构造的树。
 - c) 给出 $abbbbaa$ 的自底向上语法分析所构造的树。
11. 构造一个不包括 λ -规则的文法 G 以及串 $p \in \Sigma^+$, 这样算法 18.4.1 在尝试语法分析 p 的时候会陷入不确定的循环。
 12. 假设文法的开始符号 S 是非递归的。更改算法 18.4.1, 使得每当有串包含 S 的时候, 该算法都不会继续进行搜索。使用文法 AE 和输入 $(b + b)$ 来跟踪已更改算法的语法分析。将所得的树与图 18-3 中的搜索树相比。

参考文献注释

本章所表述的语法分析器是为了适用于特殊的应用程序而改进的图搜索算法。Knuth [1968] 中给出了图的完整展示和树遍历, 并且对于数据结构有很多的文字描述。对于语法分析和编译的一个全面的引论可以在 Aho、Sethi 和 Ullman [1986] 中找到。对于确定型语法分析技术有用的文法将会在第 19 章和第 20 章中介绍。关于语法分析的参考书目, 可以参看章节后面的参考文献注释。

第 19 章 LL(k)文法

当在搜索树中扩展一个节点的时候,有可能出现几种选项同时存在的情况,这也是第 18 章中所介绍的算法效率低下的基本原因。设 A 是句型中的最左推导,那么自顶向下的分析器将会应用每一个规则 A ,从而可以扩展其推导。对于一个给定的串来说,自底向上的分析器可能会有多个归约。但无论是哪种情况,分析器均执行所有可能的动作,将结果生成的句型加入到搜索树中去,并生成其后代。

如果存在足够的信息,能够保证每一步均可以选择唯一的动作来执行,那么这样的分析算法就是确定型的。对于自顶向下的分析器来说,这就意味着在可能的规则中,自顶向下的分析器可以确定应用其中的哪一条规则。LL(k)文法是上下文无关文法中的最大子类,该文法允许确定型的自顶向下分析器可以预读 k 个符号。这些文法是为了某些特定的分析策略而设计的,符号 LL 则描述了这样的分析策略;分析器按从左到右的方式扫描输入串,然后生成一个最左推导。预读可以读取分析器生成的那部分输入串之外的部分,这就为选择执行合适的动作提供了附加的信息。

在本章中,所有的推导和规则应用都是按最左方式进行的。这里也假设本章的文法是无二义性的,并且不包含无用符。用于检测和移除无用符的技术在 4.4 节中介绍过。

19.1 上下文无关文法中的预读

571 | 对于一个输入串 p 来说,自顶向下的分析器尝试构造该串的最左推导。分析器应用规则 A ,扩展具有形式 $S \Rightarrow uAv$ 的推导,在这里, u 是 p 的前缀。输入串中的预读可以减少必须进行检测的规则 A 的数目。如果 $p = uav$,那么终结符 a 可以通过在由分析器生成的输入串的前缀之外的部分寻找一个符号来获得。使用预读符号就可以不必考虑规则的右部是以终结符而不是以 a 开始的规则 A 。任何一个这样的规则的应用都会生成一个前缀不是 p 的终结符号串。

用下边的正则文法来思考串 $acbb$ 的推导

$$\begin{aligned} G: S &\rightarrow aS \mid cA \\ A &\rightarrow bA \mid cB \mid \lambda \\ B &\rightarrow cB \mid a \mid \lambda \end{aligned}$$

推导从开始符号 S 开始,并且预读符号 a 。这个文法包括两个 S 规则、 $S \rightarrow aS$ 和 $S \rightarrow cA$ 。很显然,应用规则 $S \rightarrow cA$ 是不能产生 $acbb$ 的推导的,这是因为 c 并不和预读符号相匹配。这样看来, $acbb$ 的推导必须从规则 $S \rightarrow aS$ 开始。

应用完规则 S 以后,预读符号就到了 c 面前。再一次可以看到,只有一个 S 规则可以产生 c 。将预读符号与每个相应规则中的终结符进行比较,则可以生成 G 中的推导的确定型的构造。

生成的前缀	预读符号	规则	推导
λ	a	$S \rightarrow aS$	$S \Rightarrow aS$
a	c	$S \rightarrow cA$	$\Rightarrow acA$
ac	b	$A \rightarrow bA$	$\Rightarrow acbA$
acb	b	$A \rightarrow bA$	$\Rightarrow acbbA$
$acbb$	λ	$A \rightarrow \lambda$	$\Rightarrow acbb$

预读一个符号对于确定型地构造文法 G 中的推导来说已经足够了。一种更为普通的方法是允许预读还未生成的输入串的部分。终结符串 p 推导的一个中间步骤具有 $S \Rightarrow uAv$ 这种形式,这里 $p = ux$ 。串 x 被称为预读串 (lookahead string) 是因为变量 A 的原因。 A 的预读集合包含了所有用于那个变量的预

读串。

定义 19.1.1 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法, 并且

i) 变量 A 的预读集合是 $LA(A)$, 其定义如下:

$$LA(A) = \{x \mid S \Rightarrow uAv \Rightarrow ux \in \Sigma^*\}.$$

ii) 对于 P 中的每个规则 $A \rightarrow w$, 规则 $A \rightarrow w$ 的预读集合均定义如下:

$$LA(A \rightarrow w) = \{x \mid wv \Rightarrow x, x \in \Sigma^*, S \Rightarrow uAv\}.$$

572

$LA(A)$ 包含了所有可以从串 Av 推导出的终结符串, 这里 uAv 是语法的左句型. $LA(A \rightarrow w)$ 是 $LA(A)$ 的子集, 子推导 $Av \Rightarrow x$ 是从规则 $A \rightarrow w$ 开始进行初始化的。

设 $A \rightarrow w_1, \dots, A \rightarrow w_n$ 是语法 G 的规则 A . 每当集合 $LA(A \rightarrow w_i)$ 分割 $LA(A)$ 的时候, 就可以利用预读串来选择合适的规则 A , 也就是说, 当集合 $LA(A \rightarrow w_i)$ 满足

i) $LA(A) = \bigcup_{i=1}^n LA(A \rightarrow w_i)$, 并且

ii) 对于所有的 $1 \leq i \leq j \leq n$ 来说, $LA(A \rightarrow w_i) \cap LA(A \rightarrow w_j) = \emptyset$.

对于任何一种上下文无关的文法来说, 第一种情况均可以得到满足; 这是直接从预读集合的定义继承下来的. 如果预读集合满足 (ii), 并且 $p = ux \in L(G)$ 是串 $p = ux$ 的部分推导, 那么 x 就必然是某个集合 $LA(A \rightarrow w_i)$ 的元素. 因此, 应用规则 $A \rightarrow w_i$ 会使得推导成功完成, 并且该规则是惟一的。

例 19.1.1 预读集合是为文法的变量和规则而构造的

$$G_1: S \rightarrow Aabd \mid cAbcd$$

$$A \rightarrow a \mid b \mid \lambda.$$

$LA(S)$ 包含了所有从 S 推导出来的终结符串. 每一个从规则 $S \rightarrow Aabd$ 推导出的终结符串都是以 a 或者 b 开头. 另一方面, 规则 $S \rightarrow cAbcd$ 开始进行初始化的推导则生成了以 c 为开始符号的串。

$$LA(S) = \{aabd, babd, abd, cabcd, cbbcd, cbcd\}$$

$$LA(S \rightarrow Aabd) = \{aabd, babd, abd\}$$

$$LA(S \rightarrow cAbcd) = \{cabcd, cbbcd, cbcd\}$$

选择合适的规则 S 的充分条件是需要获得预读串的第一个符号的信息。

为了构造变量 A 的预读集合, 就必须要考虑 G_1 中所有包含 A 的左句型的推导. 这里只有两个这样的句型, 即 $Aabd$ 和 $cAbcd$. 预读集合包含了从 $Aabd$ 和 $cAbcd$ 推导出来的终结符串。

$$LA(A \rightarrow a) = \{aabd, abcd\}$$

$$LA(A \rightarrow b) = \{babd, bbcd\}$$

$$LA(A \rightarrow \lambda) = \{abd, bcd\}$$

子串 ab 可以通过将规则 $A \rightarrow a$ 应用到串 $Abcd$, 以及将规则 $A \rightarrow \lambda$ 应用到串 $Aabd$ 来获得. 这样, 两个符号的预读对于选择正确的规则 A 来说并不是充分的. 为了区别对待这些规则, 必须要能够提供充足的信息, 这就需要能够在输入串中预读三个符号. 能够预读三个符号的自顶向下的分析器可以构造出文法 G_1 的确定型推导. \square

变量 A 的预读串是两个推导结果的连接, 一个来自于变量 A , 另一个来自于跟随 A 的句型的部分. 例 19.1.2 强调了句型的预读集合之间的依赖关系。

例 19.1.2 G_1 的预读集合从每个变量 A 、 B 和 C 至多接收一个终结符, G_2 的文法如下所示:

$$G_2: S \rightarrow ABCabcd$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow b \mid \lambda$$

$$C \rightarrow c \mid \lambda$$

G_2 中惟一包含 A 的左句型是 $ABCabcd$. 变量 B 出现在 $aBCabcd$ 和 $BCabcd$ 中, 这两者均可以通过应用规则 A 到 $ABCabcd$ 而获得. 无论是哪种情况, $BCabcd$ 都被用于构造预读集合. 类似地, 预读集合 $LA(C)$ 包含了从 $Cabcd$ 推导出的串。

$$\begin{aligned}
\text{LA}(A \rightarrow a) &= \{abcabcd, acabcd, ababcd, aabcd\} \\
\text{LA}(A \rightarrow \lambda) &= \{bcabcd, cabcd, babcd, abcd\} \\
\text{LA}(B \rightarrow b) &= \{bcabcd, babcd\} \\
\text{LA}(B \rightarrow \lambda) &= \{cabcd, abcd\} \\
\text{LA}(C \rightarrow c) &= \{cabcd\} \\
\text{LA}(C \rightarrow \lambda) &= \{abcd\}
\end{aligned}$$

对于选择规则 B 和规则 C 来说, 长度为 1 的预读可以提供充分的信息。带前缀的串 abc 可以通过使用规则 $A \rightarrow a$ 或者规则 $A \rightarrow \lambda$ 从句型 $ABCabcd$ 推导出来。长度为 4 的预读集合可以用于确定型地分析 G_1 中的串。 \square

预读集合 $\text{LA}(A)$ 和 $\text{LA}(A \rightarrow w)$ 可能包含任意长度的串。以前例子中的规则选择只是需要预读集合中的串的定长前缀。具有 k 个符号的预读集合通过截取集合 $\text{LA}(A)$ 和 $\text{LA}(A \rightarrow w)$ 中的串来获得。这里引入函数 trunc_k 来简化定长预读集合的定义。

[574] 定义 19.1.2 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法, k 为大于 0 的自然数。

i) trunc_k 是从 $\mathcal{P}(\Sigma^*)$ 到 $\mathcal{P}(\Sigma^*)$ 的函数, 其定义如下:

$$\text{trunc}_k(X) = \{u \mid u \in X \text{ 且 } \text{length}(u) \leq k \text{ 或者 } uv \in X \text{ 且 } \text{length}(u) = k\}$$

对于所有的 $X \in \mathcal{P}(\Sigma^*)$ 来说

ii) 变量 A 的长度为 k 的预读集合如下:

$$\text{LA}_k(A) = \text{trunc}_k(\text{LA}(A)).$$

iii) 规则 $A \rightarrow w$ 的长度为 k 预读集合如下所示:

$$\text{LA}_k(A \rightarrow w) = \text{trunc}_k(\text{LA}(A \rightarrow w)).$$

例 19.1.3 例 19.1.1 中用于文法 G_1 中的规则的长度为 3 的预读集合如下所示:

$$\begin{aligned}
\text{LA}_3(S \rightarrow Aabd) &= \{aab, bab, abd\} \\
\text{LA}_3(S \rightarrow cAbcd) &= \{cab, cbb, cbc\} \\
\text{LA}_3(A \rightarrow a) &= \{aab, abc\} \\
\text{LA}_3(A \rightarrow b) &= \{bab, bbc\} \\
\text{LA}_3(A \rightarrow \lambda) &= \{abd, bcd\}.
\end{aligned}$$

因为对于规则 S 或者规则 A 来说, 它们长度为 3 的预读集合中没有公共串, 所以具有三个符号的预读集合对于决定选择 G_1 中合适的规则来说是充分的。 \square

[575] 例 19.1.4 语言 $\{a^i abc^i \mid i > 0\}$ 是由文法 G_1 、 G_2 和 G_3 中的任意一个生成的。下面给出了这些文法的最短长度的预读集合, 这些预读集合对于区分可选择的生成来说是必须的。

具有一个字符的预读是不能为确定 G_1 中的规则 S 提供充分的信息的, 这是因为两个选择都是以符号 a 开头的。实际上, 需要具有三个符号的预读才能够确定合适的规则。文法 G_2 是从文法 G_1 通过使用规则 S 生成第一个 a 构造出来的。在 G_1 中, 加入变量 A , 从而生成规则 S 的右部。这种技术被称为提取左因子 (left factoring), 这是因为第一个 a 是从规则 $S \rightarrow aSc$ 和规则 $S \rightarrow aabc$ 中被提取出来的。对规则 S 使用提取左因子的方法减少了选择规则所需的预读集合的长度。

规则	预读集合
G_1 : $S \rightarrow aSc$	$\{aaa\}$
$S \rightarrow aabc$	$\{aab\}$
G_2 : $S \rightarrow aA$	$\{aa\}$
$A \rightarrow Sc$	$\{ab\}$
$A \rightarrow abc$	$\{a\}$
G_3 : $S \rightarrow aaAc$	$\{a\}$
$A \rightarrow aAc$	$\{b\}$
$A \rightarrow b$	

长度为 1 的预读对于分析 G_1 中的规则的串来说是充分的。当间接递归规则通过生成 b 而终止了其推导的时候, 递归规则 A 则生成了 a 。 \square

19.2 FIRST 集合、FOLLOW 集合和预读集合

可以看出, 利用预读集来选择合适的规则从而推导出一个所需要的串是可行的。每一个变量和规

则均生成预读集合,这对于将这个信息合并到分析器中去来说是必要的。本节将会介绍 FIRST 集和 FOLLOW 集的内容,这两种集合可以直接从文法规则中构造预读集合。

预读集合 $LA_k(A)$ 包含可以从变量 A 推导出来,并且长度最大为 k 的串的前缀。如果 A 推导出长度小于 k 的串,那么预读集合余下的部分将从以 A 为开始,并且按照文法句型进行的推导中得来。对于每一个变量 A ,这里引入集合 $FIRST_k(A)$ 和 $FOLLOW_k(A)$,从而为构造预读集合提供所需的信息。 $FIRST_k(A)$ 包含从 A 推导出来的终结字符串的前缀。 $FOLLOW_k(A)$ 包含终结字符串的前缀,而这些终结符可以放置在从 A 推导出的串的后面。为了方便,为每一个在 $(V \cup \Sigma)^*$ 中的串都定义一个 $FIRST_k$ 集合。

定义 19.2.1 设 G 是上下文无关文法。对于任意一个串 $u \in (V \cup \Sigma)^*$ 并且 $k > 0$, 集合 $FIRST_k(u)$ 的定义如下:

$$FIRST_k(u) = trunc_k(\{x \mid u \Rightarrow x, x \in \Sigma^*\}).$$

例 19.2.1 使用 G 、文法为串 S 和 ABC 建立的 $FIRST$ 集合如下 (文法 G_2 见例 19.1.2):

$$FIRST_1(ABC) = \{a, b, c, \lambda\}$$

$$FIRST_2(ABC) = \{ab, ac, bc, a, b, c, \lambda\}$$

$$FIRST_3(S) = \{abc, aca, aba, aab, bca, bab, cab\}$$

□ [576]

回想前面提到的用并置来表示将集合 X 和集合 Y 连接起来 $XY = \{xy \mid x \in X, \text{ 并且 } y \in Y\}$ 。通过使用这个概念,可以建立起 $FIRST_k$ 集合的下列关系。

引理 19.2.2 对于每一个 $k > 0$,

$$1. FIRST_k(\lambda) = \{\lambda\}$$

$$2. FIRST_k(a) = \{a\}$$

$$3. FIRST_k(av) = \{av \mid v \in FIRST_{k-1}(u)\}$$

$$4. FIRST_k(uv) = trunc_k(FIRST_k(u) FIRST_k(v))$$

$$5. \text{ 如果 } A \rightarrow w \text{ 是文法 } G \text{ 中的规则, 那么 } FIRST_k(w) \in FIRST_k(A).$$

定义 19.2.3 设 G 是上下文无关文法。对于每一个 $A \in V$, 并且 $k > 0$, 集合 $FOLLOW_k(A)$ 定义如下:

$$FOLLOW_k(A) = \{x \mid S \Rightarrow uAv, \text{ 并且 } x \in FIRST_k(v)\}.$$

$FOLLOW_k(A)$ 集合包含了终结字符串的前缀,这些终结字符串可以跟随 G 中推导的变量 A 。这是因为空串可以放在任何一个仅包含开始符号的句型的推导的后面, $\lambda \in FOLLOW_k(S)$ 。

例 19.2.2 给出变量 G_2 的长度为一和二的 FOLLOW 集合。

$$FOLLOW_1(S) = \{\lambda\}$$

$$FOLLOW_2(S) = \{\lambda\}$$

$$FOLLOW_1(A) = \{a, b, c\}$$

$$FOLLOW_2(A) = \{ab, bc, ba, ca\}$$

$$FOLLOW_1(B) = \{a, c\}$$

$$FOLLOW_2(B) = \{ca, ab\}$$

$$FOLLOW_1(C) = \{a\}$$

$$FOLLOW_2(C) = \{ab\}$$

□

在某些规则中,变量 B 出现在规则的右部,那么变量 B 的 FOLLOW 集可以从这样的规则中获得。考虑由形如 $A \rightarrow uBv$ 的规则生成的关系。 B 之后的串包括那些由 v 与其他 A 之后的终结符连接起来而生成的串。如果文法包含规则 $A \rightarrow uB$,那么任何一个 A 之后的串也可以放置在 B 之后。前面的讨论将会在引理 19.2.4 中得到总结。

引理 19.2.4 对于每一个 $k > 0$,

$$1. FOLLOW_k(S) \text{ 包含 } \lambda, \text{ 其中 } S \text{ 是文法 } G \text{ 的开始符号,}$$

$$2. \text{ 如果 } A \rightarrow uB \text{ 是文法 } G \text{ 的规则, 那么 } FOLLOW_k(A) \subseteq FOLLOW_k(B),$$

$$3. \text{ 如果 } A \rightarrow uBv \text{ 是文法 } G \text{ 的规则, 那么 } trunc_k(FIRST_k(v)FOLLOW_k(A)) \subseteq FOLLOW_k(B).$$

[577]

$FIRST_k$ 和 $FOLLOW_k$ 集合用于为文法规则构造预读集合。现在,已经定义好了长度为 k 的预读集合以及函数 $trunc_k$, 下面就来介绍定理 19.2.5。

定理 19.2.5 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。对于每一个 $k > 0$, $A \in V$, 并且规则 $A \rightarrow w = u_1 u_2 \cdots u_n$ 在 P 中,

- i) $LA_k(A) = \text{trunc}_k(\text{FIRST}_k(A) \text{ FOLLOW}_k(A))$
- ii) $LA_k(A \rightarrow w) = \text{trunc}_k(\text{FIRST}_k(w) \text{ FOLLOW}_k(A))$
 $= \text{trunc}_k(\text{FIRST}_k(u_1) \cdots \text{FIRST}_k(u_n) \text{ FOLLOW}_k(A)).$

例 19.2.3 19.1.1 中的文法如下:

$$G_1: S \rightarrow Aabd \mid cAbcd$$

$$A \rightarrow a \mid b \mid \lambda$$

该文法中用于符号的 FIRST_3 集和 FOLLOW_3 集如下:

$$\text{FIRST}_3(S) = \{aab, bab, abd, cab, cbb, cbc\}$$

$$\text{FIRST}_3(A) = \{a, b, \lambda\}$$

$$\text{FIRST}_3(a) = \{a\}$$

$$\text{FIRST}_3(b) = \{b\}$$

$$\text{FIRST}_3(c) = \{c\}$$

$$\text{FIRST}_3(d) = \{d\}$$

$$\text{FOLLOW}_3(S) = \{\lambda\}$$

$$\text{FOLLOW}_3(A) = \{abd, bcd\}.$$

集合 $LA_3(S \rightarrow Aabd)$ 是利用引理 19.2.5 中的策略由集合 $\text{FIRST}_3(A)$ 、 $\text{FIRST}_3(a)$ 、 $\text{FIRST}_3(b)$ 、 $\text{FIRST}_3(d)$ 和 $\text{FOLLOW}_3(S)$ 构造而成。

$$\begin{aligned} LA_3(S \rightarrow Aabd) &= \text{trunc}_3(\text{FIRST}_3(A) \text{ FIRST}_3(a) \text{ FIRST}_3(b) \text{ FIRST}_3(d) \text{ FOLLOW}_3(S)) \\ &= \text{trunc}_3(\{a, b, \lambda\} \{a\} \{b\} \{d\} \{\lambda\}) \\ &= \text{trunc}_3(\{aabd, babd, abd\}) \\ &= \{aab, bab, abd\} \end{aligned}$$

578 G_1 规则中长度为三的预读集合的余下部分可以在例 19.1.3 中找到。 □

19.3 强 $LL(k)$ 语法

如果 $LA(A)$ 被集合 $LA(A \rightarrow w_i)$ 分割, 那么在选择自顶向下的分析中, 预读集合可以用于选择规则 A 。本节介绍了上下文无关文法的一个子类, 我们称之为强 $LL(k)$ 文法。强 $LL(k)$ 的情况确保了预读集合 $LA_k(A)$ 是被集合 $LA_k(A \rightarrow w_i)$ 分割的。

当使用一个具有 k 个符号的预读的时候, 如果存在 k 个符号需要进行检查, 那么这是有帮助的。语言中每个串的末尾都被连接上一个边界标记 $\#$, 这样做的原因是为了保证每个预读集合都能够正确地包含 k 个符号。如果文法的开始符号 S 是间接递归的, 边界标记就可能被连接到每个规则 S 的右部。否则, 可以使用新的开始符号 S' 来对文法进行增长, 即增长规则 $S' \rightarrow S\#^k$ 。

定义 19.3.1 设 $G = (V, \Sigma, P, S)$ 是具有边界标记 $\#^k$ 的上下文无关文法。如果无论何时都会存在两个最左推导, 那么 G 是强 $LL(k)$ 的, 最左推导形式如下所示:

$$S \xRightarrow{\cdot} u_1 A v_1 \Rightarrow u_1 x v_1 \xRightarrow{\cdot} u_1 z w_1$$

$$S \xRightarrow{\cdot} u_2 A v_2 \Rightarrow u_2 y v_2 \xRightarrow{\cdot} u_2 z w_2,$$

其中 $u_i, w_i, z \in \Sigma^*$, 并且 $\text{length}(z) = k$, 那么 $x = y$ 。

下面来看下强 $LL(k)$ 文法的几个特性。首先, 在强 $LL(k)$ 文法中, 可以使用长度为 k 的预读集合确定型地分析串。

定理 19.3.2 文法 G 是强 $LL(k)$ 的, 当且仅当对于每个变量 $A \in V$, 集合 $LA_k(A \rightarrow w_i)$ 均分割 $LA_k(A)$ 。

证明: 假设对于每个变量 $A \in V$, 集合 $LA_k(A \rightarrow w_i)$ 分割 $LA_k(A)$ 。设 z 是可以通过推导获得的并且长度为 k 的终结符

$$S \Rightarrow u_1 A v_1 \Rightarrow u_1 x v_1 \Rightarrow u_1 z w_1$$

$$S \Rightarrow u_2 A v_2 \Rightarrow u_2 y v_2 \Rightarrow u_2 z w_2.$$

那么 z 不仅在 $LA_k(A \rightarrow x)$ 中, 而且在 $LA_k(A \rightarrow y)$ 中。因为集合 $LA_k(A \rightarrow w_i)$ 分割了 $LA_k(A)$, 那么 $x = y$ 并且 G 是强 LL(k)。

反过来看, 假设 G 是强 LL(k) 文法, 并且 z 是 $LA_k(A)$ 的元素。强 LL(k) 的情况能够确保只有一个规则 A , 使得该规则可以用于将 G 的句型 uAv 推导成 uzw 形式的终结字符串。因此, z 只是存在于某个确定的规则 A 的预读集合里。这就暗示着集合 $LA_k(A \rightarrow w_i)$ 分割 $LA_k(A)$ 。 ■ 579'

定理 19.3.3 如果对于某些 k 来说, G 是强 LL(k) 的, 那么 G 就是无二义性的。

直观地说, 一个可以被确定型分析的文法必须是无二义性的; 终结字符串的每一步推导只可以应用某个确定的规则。这个命题的证据留做练习。

定理 19.3.4 如果 G 具有左递归变量, 那么 G 就不是强 LL(k) 的, 这对于任何 $k > 0$ 均成立。

证明: 设 A 是左递归变量, 因为 G 不包含无用变量, 因此就存在一个终结符的推导, 该终结符包含变量 A 的左递归子推导。我们分为下面两种情况进行证明。

情况 1: A 是直接左递归。推导包括直接左递归使用形式 $A \rightarrow Ay$ 和 $A \rightarrow x$ 的规则 A , 这里 x 的第一个符号不是 A 。

$$S \Rightarrow uAv \Rightarrow uAyv \Rightarrow uxyv \Rightarrow uw \in \Sigma^*$$

长度为 k 的 w 前缀既在 $LA_k(A \rightarrow Ay)$ 中, 又在 $LA_k(A \rightarrow x)$ 中。根据定理 19.3.2, G 是强 LL(k) 的。

情况 2: A 是间接左递归的。间接递归的推导具有以下形式:

$$S \Rightarrow uAv \Rightarrow uB_1 v \Rightarrow \cdots \Rightarrow uB_n v_n \Rightarrow uAv_{n+1} \Rightarrow u x v_{n+1} \Rightarrow uw \in \Sigma^*.$$

再一次可以看出, 因为集合 $LA_k(A \rightarrow B_1 v)$ 和 $LA_k(A \rightarrow x)$ 是相交的, 所以 G 是 LL(k) 的。 ■

19.4 FIRST_k 集合的构造

下面介绍一个算法, 这个算法可以为带有边界标记[#]的上下文无关文法构造出长度为 k 的预读集合。这是通过为文法的变量生成 FIRST_k 集合和 FOLLOW_k 集合来完成的。那么预读集合就可以通过使用定理 19.2.5 中的技术来构造。

构造预读集合的初始化步骤从生成 FIRST_k 集合开始。考虑具有形式 $A \rightarrow u_1 u_2 \cdots u_n$ 的规则。这个规则生成的 FIRST_k(A) 的子集可以从集合 FIRST_k(u_1), FIRST_k(u_2), \cdots , FIRST_k(u_i) 以及 FOLLOW_k(A) 中构造出来。为串构造 FIRST_k 集合就可以看成是寻找串中变量的集合。 580

算法 19.4.1

FIRST_k 集合的构造

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

1. for 每个 $a \in \Sigma$ do $F'(a) := \{a\}$

2. for 每个 $A \in V$ do $F(A) := \begin{cases} \{\lambda\} & \text{if } A \rightarrow \lambda \text{ 是 } P \text{ 中的规则} \\ \emptyset & \text{否则} \end{cases}$

3. repeat

3.1 for 每个 $A \in V$ do $F'(A) := F(A)$

3.2 for 每个规则 $A \rightarrow u_1 u_2 \cdots u_n (n > 0)$ do

$$F(A) := F(A) \cup \text{trunc}_k(F'(u_1)F'(u_2)\cdots F'(u_n))$$

until 对于所有 $A \in V$ 都有 $F(A) = F'(A)$ 成立

4. FIRST_k(A) = $F(A)$

$\text{FIRST}_k(A)$ 的元素在步骤 3.2 中生成。在 repeat - until 循环的每个迭代的开始, 辅助集合 $F'(A)$ 即被指派成 $F(A)$ 的当前的值。从连接 $F'(u_1)F'(u_2)\cdots F'(u_n)$ 获得的串, 则被加入到 $F(A)$ 中去, 这里 $A \rightarrow u_1u_2\cdots u_n$ 是文法 G 中的规则。当某个迭代中 $F(A)$ 已经没有任何集合需要被更改, 则算法停止。

例 19.4.1 算法 19.4.1 为文法变量构造 FIRST_2 集, 文法定义如下所示:

$$\begin{aligned} G: S &\rightarrow A\#\# \\ A &\rightarrow aAd \mid BC \\ B &\rightarrow bBc \mid \lambda \\ C &\rightarrow acC \mid ad. \end{aligned}$$

对于每一个 $a \in \Sigma$, 集合 $F'(a)$ 被初始化为 $\{a\}$ 。repeat - until 循环的动作利用语法规则的右部进行描述。步骤 3.2 生成了 G 规则的指定声明。

$$\begin{aligned} F(S) &:= F(S) \cup \text{trunc}_2(F'(A)\{\#\}\{\#\}) \\ F(A) &:= F(A) \cup \text{trunc}_2(\{a\}F'(A)\{d\}) \cup \text{trunc}_2(F'(B)F'(C)) \\ F(B) &:= F(B) \cup \text{trunc}_2(\{b\}F'(B)\{c\}) \\ F(C) &:= F(C) \cup \text{trunc}_2(\{a\}\{c\}F'(C)) \cup \text{trunc}_2(\{a\}\{d\}) \end{aligned}$$

在循环的每个迭代之后给出集合 $F(S)$, $F(A)$, $F(B)$ 和 $F(C)$ 的状态, 这样可以追踪 FIRST_k 集的生成情况。回想前面说到的空集合和生成空集合的集合之间的连接。

	$F(S)$	$F(A)$	$F(B)$	$F(C)$
0	\emptyset	\emptyset	$\{\lambda\}$	\emptyset
1	\emptyset	\emptyset	$\{\lambda, bc\}$	$\{ad\}$
2	\emptyset	$\{ad, bc\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
3	$\{ab, bc\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
4	$\{ad, bc, aa, ab, bb, ac\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
5	ad, bc, aa, ab, bb, ac	ad, bc, aa, ab, bb, ac	λ, bc, bb	ad, ac

定理 19.4.2 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。对于每个变量 $A \in V$, 算法 19.4.1 可以生成其 $\text{FIRST}_k(A)$ 集合。

证明: 证明包括说明步骤 3 中的 repeat - until 循环会终止, 并且终止的时候, $F(A) = \text{FIRST}_k(A)$ 。

i) 算法 19.4.1 终止的情况。repeat - until 循环的迭代次数会受到限制, 这是因为对于长度为 k 或者长度小于 k 的预读串来说, 其数目是受到限制的。

ii) $F(A) = \text{FIRST}_k(A)$ 的情况。首先, 需要证明对于任意变量 $A \in V, F(A) \subseteq \text{FIRST}_k(A)$ 均成立。为了完成这个证明, 就要先说明在每一个 repeat - until 循环的迭代的开始, $F(A) \subseteq \text{FIRST}_k(A)$ 。经过检查, 这个式子在第一个迭代之前是有效的。假设在循环的第 m 个迭代之前 (包括第 m 个迭代), 所有变量 A 均满足这个式子。

那么, 在第 $m+1$ 个迭代的过程中, 对 $F(A)$ 的惟一增加来自于下述形式的指派声明:

$$F(A) := F(A) \cup \text{trunc}_k(F'(u_1)F'(u_2)\cdots F'(u_n)),$$

这里 $A \rightarrow u_1u_2\cdots u_n$ 是文法 G 的规则。根据归纳假设, 每一个 $F'(u_i)$ 集合都是 $\text{FIRST}_k(u_i)$ 的子集。如果在这个迭代中, u 被加入到 $F(A)$ 中去, 那么

$$\begin{aligned} u &\in \text{trunc}_k(F'(u_1)F'(u_2)\cdots F'(u_n)) \\ &\subseteq \text{trunc}_k(\text{FIRST}_k(u_1)\text{FIRST}_k(u_2)\cdots \text{FIRST}_k(u_n)) \\ &= \text{FIRST}_k(u_1u_2\cdots u_n) \\ &\subseteq \text{FIRST}_k(A) \end{aligned}$$

并且 $u \in \text{FIRST}_k(A)$ 。最后两步推导过程是从引理 19.2.2 中推导出来的。

现在要说明的是, 在循环完成的时候, $F(A) \subseteq \text{FIRST}_k(A)$ 是成立的。设 $F_m(A)$ 是集合 $F(A)$ 在 m 次循环反复之后的值。假设 repeat - until 循环在 j 次反复之后停止。首先, 我们看一下对于某个 $m > j$, 是否存在一个串, 当该串可以在 $F_m(A)$ 中的时候, 该串就在 $F(A)$ 中。这是因为对于任何循环迭代次

数超过 j 的, 集合 $F(A)$ 和 $F'(A)$ 是不可能一致的。这里将要说明 $FIRST_i(A) \subseteq F(A)$ 是成立的。

设 x 是 $FIRST_i(A)$ 中的一个串。那么存在一个推导 $A \xRightarrow{m} w$, 这里 $w \in \Sigma^*$, 并且 x 是长度为 k 的 w 前缀。根据推导的归纳长度, 可以得到 $x \in F_m(A)$ 。这里主要包含了可以从一条规则推导出来的终结符串。如果 $A \rightarrow w \in P$, 那么 x 就被加入到 $F_1(A)$ 中。

假设对于 V 中所有变量 A , $trunc_i(w \mid A \xRightarrow{m} w \in \Sigma^*)$ 均成立。设 $x \in trunc_i(w \mid A \xRightarrow{m+1} w \in \Sigma^*)$; 也就是说, x 是通过应用 $m+1$ 规则从 A 推导出来的终结符串的前缀。这里将会证明 $x \in F_{m+1}(A)$ 。 w 的推导可以写成如下形式:

$$A \Rightarrow u_1 u_2 \cdots u_n \xRightarrow{m} x_1 x_2 \cdots x_n = w,$$

其中, $u_i \in V \cup \Sigma$ 并且 $u_i \Rightarrow x_i$ 。很明显, 每一个子推导 $u_i \Rightarrow x_i$ 的长度均小于 $m+1$ 。根据归纳假设, 通过按长度 k 来截取 x_i 所获得的串在 $F_m(u_i)$ 中。

在第 $m+1$ 个迭代中, $F_{m+1}(A)$ 因为下述集合而得到增长:

$$trunc_k(F'_{m+1}(u_1) \cdots F'_{m+1}(u_n)) = trunc_k(F_m(u_1) \cdots F_m(u_n))$$

因此,

$$\{x\} = trunc_k(x_1 x_2 \cdots x_n) \subseteq trunc_k(F_m(u_1) \cdots F_m(u_n))$$

并且 x 是 $F_{m+1}(A)$ 中的元素。所有在 $FIRST_k(A)$ 中的串均在 $F_j(A)$ 中, 得证。 ■

19.5 FOLLOW_k 集合的构造

引理 19.2.4 中的结论来自于生成 FOLLOW_k 集合的算法的基础。FOLLOW_k 集合是从 FIRST_k 集合以及一些规则构造出来的, 其中, 这些规则必须是那些 A 出现在其右边的规则。算法 19.5.1 通过使用辅助集合 $FL(A)$ 生成 FOLLOW_k(A) 集合。集合 $FL'(A)$ 触发停止状态, 并且维护前一个反复中指派给 $FL(A)$ 的值。

算法 19.5.1

集合 FOLLOW_k 的构造

输入: 上下文无关文法 $G = (V, \Sigma, P, S)$

每个 $A \in V$ 的 $FIRST_k(A)$

1. $FL(S) := \{\lambda\}$

2. for 每个 $A \in V - \{S\}$ do $FL(A) := \emptyset$

3. repeat

3.1 for 每个 $A \in V$ do $FL'(A) := FL(A)$

3.2 for 每个规则 $A \rightarrow w = u_1 u_2 \cdots u_n$ ($w \notin \Sigma^*$) do

3.2.1 $L := FL'(A)$

3.2.2 if $u_n \in V$ then $FL(u_n) := FL(u_n) \cup L$

3.2.3 for $i := n-1$ to 1 do

3.2.3.1 $L := trunc_k(FIRST_k(u_{i+1})L)$

3.2.3.2 if $u_i \in V$ then $FL(u_i) := FL(u_i) \cup L$

end for

end for

until 对于任何 $A \in V$ 都有 $FL(A) = FL'(A)$ 成立

4. FOLLOW_k(A): = $FL(A)$

583

加入到声明 3.2.2 和 3.2.2.2 中的 $FL(A)$ 里的每个元素均在 FOLLOW_k 中, 这就使得式子 $FL(A) \subseteq FOLLOW_k(A)$ 可以因此而得到确定。相反的包含关系可以通过证明 FOLLOW_k 中的每个元素均被加入到 $FL(A)$ 中, 并且 $FL(A)$ 在 repeat-until 循环的终结符前面来获得。细节问题留做练习。

例 19.5.1 算法 19.5.1 用于为例 19.4.1 中的文法 G 的每个变量构造集合 FOLLOW_2 。repeat-until 循环的内部按照从右到左的方式处理每条规则。循环的动作是通过指派声明来获得的，这里的指派声明来自于语法的规则。

规则	指派
$S \rightarrow A\#\#$	$\text{FL}(A) := \text{FL}(A) \cup \text{trunc}_2(\{\#\#\} \text{FL}'(S))$
$A \rightarrow aAd$	$\text{FL}(A) := \text{FL}(A) \cup \text{trunc}_2(\{d\} \text{FL}'(A))$
$A \rightarrow BC$	$\text{FL}(C) := \text{FL}(C) \cup \text{FL}'(A)$ $\text{FL}(B) := \text{FL}(B) \cup \text{trunc}_2(\text{FIRST}_2(C) \text{FL}'(A))$ $= \text{FL}(B) \cup \text{trunc}_2(\{ad, ac\} \text{FL}'(A))$
$B \rightarrow bBc$	$\text{FL}(B) := \text{FL}(B) \cup \text{trunc}_2(\{c\} \text{FL}'(B))$

规则 $C \rightarrow acC$ 在列表中被忽略了，这是因为这条规则产生的指派是 $\text{FL}(C) := \text{FL}(C) \cup \text{FL}'(C)$ 。跟踪算法 19.5.1 的生成结果显示如下：

	$\text{FL}(S)$	$\text{FL}(A)$	$\text{FL}(B)$	$\text{FL}(C)$
0	$\{\lambda\}$	\emptyset	\emptyset	\emptyset
1	$\{\lambda\}$	$\{\#\#\}$	\emptyset	\emptyset
2	$\{\lambda\}$	$\{\#\#, d\#\}$	$\{ad, ac\}$	$\{\#\#\}$
3	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca\}$	$\{\#\#, d\#\}$
4	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca, cc\}$	$\{\#\#, d\#, dd\}$
5	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca, cc\}$	$\{\#\#, d\#, dd\}$

584

例 19.5.2 用于文法 G 的规则，且长度为二的预读集合是从 FIRST_2 集合和 FOLLOW_2 集合构造出来的，这两个集合是由例 19.4.1 和例 19.5.1 生成的。

$$\text{LA}_2(S \rightarrow A\#\#) = \{ad, bc, aa, ab, bb, ac\}$$

$$\text{LA}_2(A \rightarrow aAd) = \{aa, ab\}$$

$$\text{LA}_2(A \rightarrow BC) = \{bc, bb, ad, ac\}$$

$$\text{LA}_2(B \rightarrow bBc) = \{bb, bc\}$$

$$\text{LA}_2(B \rightarrow \lambda) = \{ad, ac, ca, cc\}$$

$$\text{LA}_2(C \rightarrow acC) = \{ac\}$$

$$\text{LA}_2(C \rightarrow ad) = \{ad\}$$

G 是强 $\text{LL}(2)$ 的，这是因为预读集合与每对可选择的规则是不相交的。

前面讲述的算法为确定一个文法是否是强 $\text{LL}(k)$ 的提供了判定步骤。上述算法从使用算法 19.4.1 和算法 19.5.1 生成 FIRST_k 集合和 FOLLOW_k 集合开始。定理 19.2.5 中介绍的技术就是用于构造长度为 k 的预读集合的。根据定理 19.3.2，当且仅当集合 $\text{LA}_k(A \rightarrow x)$ 和 $\text{LA}_k(A \rightarrow y)$ 与不同的规则 A 的每一对均不相交，文法才是强 $\text{LL}(k)$ 的。

19.6 强 $\text{LL}(1)$ 文法

文法 AE 在 18.1 节中介绍过，该文法用于生成包含惟一变量 b 的中缀加法表达式。 AE 不是强 $\text{LL}(k)$ 的，因为它包含了直接左递归规则 A 。本节将会对 AE 做出修改，使得其变成强 $\text{LL}(1)$ 文法，这样的文法可以生成加法表达式。为了保证结果文法是强 $\text{LL}(1)$ 的，对每个规则都需要建立起长度为一的预读集合。

转换是从增加边界标记 $\#$ 到 AE 生成的串上开始的。这就确保了预读集合不会包含空串，文法

$$AE: S \rightarrow A\#$$

$$A \rightarrow T$$

$$A \rightarrow A + T$$

$$T \rightarrow b$$

$$T \rightarrow (A)$$

585

生成了 $L(AE)$ 中的串, 并加上边界标记 #。直接左递归可以通过使用 4.5 节中介绍的技术来解决。变量 Z 用来将左递归转变成为右递归, 生成相应的文法 AE_1 。

$$AE_1: S \rightarrow A\#$$

$$A \rightarrow T$$

$$A \rightarrow TZ$$

$$Z \rightarrow +T$$

$$Z \rightarrow +TZ$$

$$T \rightarrow b$$

$$T \rightarrow (A)$$

AE_1 仍然不能是强 LL(1) 的, 这是因为两个规则 A 都将 T 作为其右部的第一个符号。这种困难是通过使用新的变量 B 左提取规则 A 来解决的。相似地, 规则 Z 的右部以相同的子串开始。变量 Y 通过提取规则 Z 来引入。 AE_2 则是通过对 AE_1 的更改得到的, AE_2 的文法显示如下:

$$AE_2: S \rightarrow A\#$$

$$A \rightarrow TB$$

$$B \rightarrow Z$$

$$B \rightarrow \lambda$$

$$Z \rightarrow +TY$$

$$Y \rightarrow Z$$

$$Y \rightarrow \lambda$$

$$T \rightarrow b$$

$$T \rightarrow (A)$$

为了说明 AE_2 是强 LL(1) 的, 那么, 用于文法的变量, 且长度为 1 的预读集合必须要能够满足定理 19.3.2 的分割情况。这里从跟踪算法 19.4.1 的集合序列开始, 其中算法 19.4.1 构造了 $FIRST_1$ 集合, 显示如下:

	$F(S)$	$F(A)$	$F(B)$	$F(Z)$	$F(Y)$	$F(T)$
0	\emptyset	\emptyset	$\{\lambda\}$	\emptyset	$\{\lambda\}$	\emptyset
1	\emptyset	\emptyset	$\{\lambda\}$	$\{+\}$	$\{\lambda\}$	$\{b, ($
2	\emptyset	$\{b, ($	$\{\lambda, +$	$\{+\}$	$\{\lambda, +$	$\{b, ($
3	$\{b, ($	$\{b, ($	$\{\lambda, +$	$\{+\}$	$\{\lambda, +$	$\{b, ($
4	$\{b, ($	$\{b, ($	$\{\lambda, +$	$\{+\}$	$\{\lambda, +$	$\{b, ($

586

类似地, $FOLLOW_2$ 集合是使用算法 19.5.1 生成的, 该集合显示如下:

	$FL(S)$	$FL(A)$	$FL(B)$	$FL(Z)$	$FL(Y)$	$FL(T)$
0	$\{\lambda\}$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
1	$\{\lambda\}$	$\{\#,)\}$	\emptyset	\emptyset	\emptyset	\emptyset
2	$\{\lambda\}$	$\{\#,)\}$	$\{\#,)\}$	\emptyset	\emptyset	\emptyset
3	$\{\lambda\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	\emptyset	\emptyset
4	$\{\lambda\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	\emptyset
5	$\{\lambda\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$
6	$\{\lambda\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$	$\{\#,)\}$

长度为 1 的预读集合是从 $FIRST_1$ 集合和 $FOLLOW_1$ 集合得到的, 结果显示如下:

$$LA_1(S \rightarrow A\#) = \{b, (\}$$

$$LA_1(A \rightarrow TB) = \{b, (\}$$

$$LA_1(B \rightarrow Z) = \{+\}$$

$$LA_1(B \rightarrow \lambda) = \{\#,)\}$$

$$LA_1(Z \rightarrow +TY) = \{+\}$$

$$LA_1(Y \rightarrow Z) = \{+\}$$

$$LA_1(Y \rightarrow \lambda) = \{\#,)\}$$

$$LA_1(T \rightarrow b) = \{b\}$$

$$LA_1(T \rightarrow (A)) = \{(\}$$

因为可选择规则的预读集合是不相交的, 所以文法 AE_2 是 $LL(1)$ 的。

19.7 强 $LL(k)$ 分析器

利用强 $LL(k)$ 文法进行分析, 需要从为文法的每个规则构造其预读集合开始。一旦这些集合创建完成, 它们就可以用于分析任何长度的串。用于分析强 $LL(k)$ 文法的策略将在算法 19.7.1 中介绍, 该算法包括一个循环, 此循环将预读串和预读集合进行比较, 并且使用合适的规则。

算法 18.2.1 中给出了自顶向下分析器中的多种多样的规则, 与这些规则的检查不同的是, 使用强 $LL(k)$ 文法进行节点扩张是受到限制的, 也就是说, 这种情况最多只能应用一种规则。预读串和预读集合为消除无需考虑的规则提供了充分的信息。

算法 19.7.1

用于强 $LL(k)$ 文法的确定型分析器

输入: 强 $LL(k)$ 文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

P 中每个规则的向前集合 $LA_k(A \rightarrow w)$

1. $q := S$ (q is the sentential form to be expanded)

2. repeat

 设 $q = uAv$, 其中 A 是 q 最左边的变量,

 设 $p = uyz$, 其中 $length(y) = k$.

 2.1. 对于某个 A 规则有 if $y \in LA_k(A \rightarrow w)$ then $q := uwv$

 until $q = p$ or 对所有 A 规则有 $y \notin LA_k(A \rightarrow w)$

3. if $q = p$ then 接收 else 拒绝接收

文法中, 边界标记的存在确保了预读串 y 中包含了 k 个符号, 无论何时, 只要预读串不是预读集合中的一个元素, 那么这个输入串就会被否决。当预读串在 $LA_k(A \rightarrow w)$ 中的时候, 一个新的句型被构造出来, 这是通过应用规则 $A \rightarrow w$ 到当前串 uAv 来实现的。如果这个规则的应用生成了输入串, 那么这个输入就会被接收。否则, 为了得到句型 uwv , 循环就这样反复地进行。

例 19.7.1 利用算法 19.7.1 和 19.6 节中介绍的强 $LL(k)$ 文法 AE_2 的预读集合来分析串 $(b + b)\#$ 。下表中的每一行均表示算法 19.7.1 中步骤 2 的迭代。

u	A	v	预读	规则	推导
λ	S	λ	$($	$S \rightarrow A\#$	$S \Rightarrow A\#$
λ	A	$\#$	$($	$A \rightarrow TB$	$\Rightarrow TB\#$
λ	T	$B\#$	$($	$T \rightarrow (A)$	$\Rightarrow (A)B\#$
$($	A	$)B\#$	b	$A \rightarrow TB$	$\Rightarrow (TB)B\#$
$($	T	$B)B\#$	b	$T \rightarrow b$	$\Rightarrow (bB)B\#$

(续)

u	A	v	预读	规则	推导
$(b$	B	$)B\#$	$+$	$B \rightarrow Z$	$\Rightarrow (bZ)B\#$
$(b$	Z	$)B\#$	$+$	$Z \rightarrow +TY$	$\Rightarrow (b+TY)B\#$
$(b+$	T	$)Y)B\#$	b	$T \rightarrow b$	$\Rightarrow (b+bY)B\#$
$(b+b$	Y	$)B\#$	$)$	$Y \rightarrow \lambda$	$\Rightarrow (b+b)B\#$
$(b+b)$	B	$\#$	$\#$	$B \rightarrow \lambda$	$\Rightarrow (b+b)\#$

□ [588]

19.8 LL(k)文法

强 LL(k) 文法中的预读集合为选择规则提供了一个全局的标准。如果 A 是句型中的最左变量，并且这个句型正在通过分析而得到扩展，那么分析器生成的预读串和预读集合为选择合适的规则 A 提供了充分的信息。这种选择并不依赖于包含 A 的句型。LL(k) 文法提供了一个局部选择标准：选择哪条规则是由预读集合和句型这两者决定的。

定义 19.8.1 设 $G = (V, \Sigma, P, S)$ 是带有边界标记 $\#$ 的上下文无关文法。如果无论何时必然存在两个如下所示的最左推导

$$S \xRightarrow{\cdot} uAv \Rightarrow uxv \xRightarrow{\cdot} uzv_1$$

$$S \xRightarrow{\cdot} uAv \xRightarrow{\cdot} uyv \xRightarrow{\cdot} uzv_2,$$

其中 $u, w_i, z \in \Sigma^*$ 并且 $\text{length}(z) = k$ 。那么 $x = y$ ，则 G 是 LL(k)。

注意定义 19.3.1 和定义 19.8.1 中的推导的区别。强 LL(k) 的情况要求具有唯一的规则 A ，可以从任何包含 A 的句型中推导出预读串。LL(k) 文法只是要求用于固定句型 uAv 的规则是唯一的。必须为每个句型都定义 LL(k) 文法的预读集合。

定义 19.8.2 设 $G = (V, \Sigma, P, S)$ 是带有末端标记 $\#$ 的上下文无关文法，并且 uAv 是 G 的一个句型。

i) 句型 uAv 的预读集合是由 $LA_k(uAv) = \text{FIRST}_k(Av)$ 定义的。

ii) 句型 uAv 的预读集合以及规则 $A \rightarrow w$ 是由 $LA_k(uAv, A \rightarrow w) = \text{FIRST}_k(wv)$ 定义的。

上述与定理 19.3.2 相似的结论可以用于建立 LL(k) 文法。对用于句型 uAv 的规则的惟一选择标准是要求集合 $LA_k(uAv)$ 被规则 A 生成的预读集合 $LA_k(uAv, A \rightarrow w)$ 分割。如果文法是强 LL(k) 的，那么分割就能得到保证，并且语法也是 LL(k) 的。

例 19.8.1 LL(k) 文法并不一定是强 LL(k) 的。考虑如下文法：

$$G_1: S \rightarrow Aabd \mid cAbcd$$

$$A \rightarrow a \mid b \mid \lambda$$

[589]

该文法的预读集合在例 19.4.1 中给出。 G_1 是强 LL(3) 的，但不是强 LL(2) 的，这是因为串 ab 既在 $LA_2(A \rightarrow a)$ 中，也在 $LA_2(A \rightarrow \lambda)$ 中。用于包含变量 S 和句型 A ，并且长度为二的预读集合显示如下：

$$LA_2(S, S \rightarrow Aabd) = \{aa, ba, ab\}$$

$$LA_2(S, S \rightarrow cAbcd) = \{ca, cb\}$$

$$LA_2(Aabd, A \rightarrow a) = \{aa\}$$

$$LA_2(cAbcd, A \rightarrow a) = \{ab\}$$

$$LA_2(Aabd, A \rightarrow b) = \{ba\}$$

$$LA_2(cAbcd, A \rightarrow b) = \{bb\}$$

$$LA_2(Aabd, A \rightarrow \lambda) = \{ab\}$$

$$LA_2(cAbcd, A \rightarrow \lambda) = \{bc\}.$$

文法是 LL(2) 的，这是因为给定句型的可选项是不相交的。 □

例 19.8.2 二个符号的预读对于文法规则的局部选择来说是充分的，文法显示如下：

$$G: S \rightarrow aBA d \mid bBbAd$$

$$A \rightarrow abA \mid c$$

$$B \rightarrow ab \mid a.$$

规则 S 和 A 可以使用具有二个符号的预读来选择；所以这里可以将注意力转换到选择规则 B 上去。规

则 B 的预读集合显示如下:

$$LA_3(aBAd, B \rightarrow ab) = \{aba, abc\}$$

$$LA_3(aBAd, B \rightarrow a) = \{aab, acd\}$$

$$LA_3(bBbAd, B \rightarrow ab) = \{abb\}$$

$$LA_3(bBbAd, B \rightarrow a) = \{aba, abc\}.$$

B 规则将包含 B 的那两个句型的预读集合进行分割, 且该预读集合的长度为 3。因此, G 是 $LL(3)$ 的。强 $LL(k)$ 的情况可以通过检查规则 B 的预读集合来核对。

$$LA(B \rightarrow ab) = ab(ab)^*cd \cup abb(ab)^*cd$$

$$LA(B \rightarrow a) = a(ab)^*cd \cup ab(ab)^*cd$$

对于任何一个整数 k , 存在一个长度大于 k 的串, 且该串既在 $LA(B \rightarrow ab)$ 中, 又在 $LA(B \rightarrow a)$ 中。因此, 对于任何 k 来说, G 都不是强 $LL(k)$ 的。□

590 使用 $LL(k)$ 文法进行确定型分析要求为分析过程中生成的句型构造局部预读集合。句型的预读集合可以从变量的 $FIRST_k$ 集合和文法终结符处直接构造。给出预读集合 $LA_k(uAv, A \rightarrow w)$ 如下, 这里 $w = w_1 \cdots w_n$, 并且 $v = v_1 \cdots v_m$, 如下所示:

$$trunc_k(FIRST_k(w_1) \cdots FIRST_k(w_n) FIRST_k(v_1) \cdots FIRST_k(v_m)).$$

用于 $LL(k)$ 文法的分析算法可以从算法 19.7.1 中获得, 这是通过将局部预读集合加入到构造中来实现的。

算法 19.8.3

用于 $LL(k)$ 文法的确定型分析

输入: $LL(k)$ 文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

任何 $A \in V$ 的 $FIRST_k(A)$

1. $q := S$

2. repeat

 设 $q = uAv$, 其中 A 是 q 中最左边的变量, 且

 设 $p = uyz$, 其中 $length(y) = k$.

 2.1. for 为每个规则 $A \rightarrow w$ 构造集合 $LA_k(uAv, A \rightarrow w)$

 2.2. if 对于某个规则 A 有 $y \in LA_k(uAv, A \rightarrow w)$ 成立 then $q := uwv$

 until $q = p$ or 对于所有 A 都有 $y \notin LA_k(uAv, A \rightarrow w)$

3. if $q = p$ then 接收 else 拒绝

强 $LL(k)$ 文法家族是 $LL(k)$ 的真子集。局部预读集合在选择合适规则的问题上允许使用更多上下文有关的信息。强 $LL(k)$ 文法包含了变量 A 和预读串, 用以决定要将哪条规则应用于句型 uAv 。在 $LL(k)$ 文法中, 分析器生成的终结符前缀 u 也可能用于规则选择。强 $LL(k)$ 文法并不能产生所有可以确定型分析的上下文无关语言。练习 14 给出了一个语言的例子, 这种语言可以通过确定型下推自动机来分析, 但该语言并不是由 $LL(k)$ 文法生成的。

19.9 练习

1. 设 G 是上下文无关文法, 开始符号为 S 。证明 $LA(S) = L(G)$ 。

2. 给出下列文法中每个变量的预读集合。

a) $S \rightarrow ABab \mid bAcc$

b) $S \rightarrow aS \mid A$

$A \rightarrow a \mid c$

$A \rightarrow ab \mid b$

$B \rightarrow b \mid c \mid \lambda$

- c) $S \rightarrow AB \mid ab$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow bB \mid \lambda$
- d) $S \rightarrow aAbBc$
 $A \rightarrow aA \mid cA \mid \lambda$
 $B \rightarrow bBc \mid bc$

3. 给出下列文法中每个变量的 $FIRST_k$ 集合和 $FOLLOW_k$ 集合, 哪些文法是 LL(1) 的?

- a) $S \rightarrow aAB\#$
 $A \rightarrow a \mid \lambda$
 $B \rightarrow b \mid \lambda$
- b) $S \rightarrow AB\#$
 $A \rightarrow aAb \mid B$
 $B \rightarrow aBc \mid \lambda$
- c) $S \rightarrow ABC\#$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow bBc \mid \lambda$
 $C \rightarrow cA \mid dB \mid \lambda$
- d) $S \rightarrow aAd\#$
 $A \rightarrow BCD$
 $B \rightarrow bB \mid \lambda$
 $C \rightarrow cC \mid \lambda$
 $D \rightarrow bD \mid \lambda$

4. 给出可以生成下述语言的强 LL(1) 文法。

- a) $\{a^i b^j c^i \mid i \geq 0, j \geq 0\}$
b) $\{a^i b^j c \mid i \geq 1, j \geq 0\}$

5. 证明文法

$$S \rightarrow aSa \mid bSb \mid \lambda$$

是强 LL(2) 的, 但不是强 LL(1) 的。

6. 利用算法 19.4.1 和算法 19.5.1 为下述文法的变量构造 $FIRST_k$ 集和 $FOLLOW_k$ 集。这些文法是强 LL(2) 的吗?

- a) $S \rightarrow ABC\#\#$
 $A \rightarrow aA \mid a$
 $B \rightarrow bB \mid \lambda$
 $C \rightarrow cC \mid a \mid b \mid c$
- b) $S \rightarrow A\#\#$
 $A \rightarrow bBA \mid BcAa \mid \lambda$
 $B \rightarrow acB \mid b$

7. 证明引理 19.2.2 中的 3、4 和 5 部分。

8. 证明定理 19.3.3。

9. 证明下述文法对于任何 k 来说都不是强 LL(k) 的。构造一个可以由该语法生成的语言的确定型 PDA。

- a) $S \rightarrow aSb \mid A$
 $A \rightarrow aAc \mid \lambda$
- b) $S \rightarrow A \mid B$
 $A \rightarrow aAb \mid ab$
 $B \rightarrow aBc \mid ac$

- c) $S \rightarrow A$
 $A \rightarrow aAb \mid B$
 $B \rightarrow aB \mid a$

10. 证明算法 19.5.1 可以生成 $FOLLOW_k(A)$ 集。

11. 对下述给出的文法进行修改, 使得其可以称为等价的强 LL(1) 文法。建立预读集合, 以保证修改后的语法是强 LL(1) 的。

- a) $S \rightarrow A\#$
 $A \rightarrow aB \mid Ab \mid Ac$
 $B \rightarrow bBc \mid \lambda$
- b) $S \rightarrow aA\# \mid abB\# \mid abcC\#$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow bB \mid \lambda$
 $C \rightarrow cC \mid \lambda$

12. 利用 LL(1) 分析和 AE_2 文法分析下述串。使用例 19.7.1 的格式来跟踪分析的动作。 AE_2 的预读集合在 19.6 节中给出。

- a) $b + (b)\#$
b) $((b))\#$
c) $b + b + b\#$
d) $b. + + b\#$

13. 为文法规则构造预读集合。若使文法是强 $LL(k)$ 的, 那么最小 k 值是多少? 为每个句型合并和规则构造预读集合。若使文法是 $LL(k)$ 的, 那么最小的 k 值是多少?

$$a) S \rightarrow aAcaa \mid bAbcc$$

$$A \rightarrow a \mid ab \mid \lambda$$

$$b) S \rightarrow aAbc \mid bABbd$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow a \mid b$$

$$c) S \rightarrow aAbB \mid bAbA$$

$$A \rightarrow ab \mid a$$

$$B \rightarrow aB \mid b$$

14. 证明不存在 $LL(k)$ 文法, 使其可以生成下述语言

$$L = \{a^i \mid i \geq 0\} \cup \{a^i b^i \mid i \geq 0\}.$$

设计一个可以接收 L 的确定型下推自动机。

15. 证明文法是强 $LL(1)$ 的, 当且仅当它是 $LL(1)$ 的。

16. 证明上下文无关文法 G 是 $LL(k)$ 的, 当且仅当, 对于每个句型 uAv 来说, 预读集合 $LA_k(uAv)$ 均被集合 $LA_k(uAv, A \rightarrow w_i)$ 分割。

参考文献注释

使用 $LL(k)$ 文法进行分析是由 Lewis 和 Stearns (1968) 首先提出来的。Rosenkrantz 和 Stearns (1970) 进一步发展了 $LL(k)$ 文法和确定型分析。Aho 和 Ullman (1973) 又检验了 $LL(k)$ 语言和其他类能够被确定型分析的语言之间的联系。Kuri Suono (1969) 提出了 $LL(k)$ 分层。Foster (1968)、Wood (1969)、Stearns (1971) 以及 Soisalon-Soininen 和 Ukkonen (1979) 引入更改文法的技术, 使得更改后的文法可以满足 $LL(k)$ 或者强 $LL(k)$ 的环境。

为由 $LL(1)$ 文法定义的语言建立编译器通常都会使用递归方法。这种方法允许机器代码的生成附带语法分析。关于语法分析的一个全方位的引论和编译可以在 Aho、Sethi 和 Ullman (1986) 中找到。

第 20 章 LR(k)文法

自底向上的分析器生成了移进和归约序列, 这样可以将输入串归约到文法的开始符号。确定型的分析器必须要能够在此过程中合成附加的信息, 以便于在有多种可能性共同存在的情况下做出正确的选择。如果一个具有 k 个符号的预读集合可以提供足够的信息以便于做出选择, 那么这个文法就是 LR(k) 文法。LR 表示这些串是按照从左到右的方式进行分析的, 这样就构成了一个最右推导。LR(k) 文法在理论上是很重要的意义的, 这是因为对于每一种可以被确定型分析的上下文无关语言来说, 按照从左到右的方式读入一个输入串是由 LR(k) 文法生成。而对于应用上的意义来说, LR 方式可以为自底向上的分析发生器提供基础, 程序将会自动地直接从文法规则生成分析器。

本章中所有的推导都是最右推导。这里假设文法的开始符号是非递归的, 并且文法里所有的符号都是有用的。

20.1 LR(0)上下文

确定型的自底向上分析器尝试将输入串归约成文法的开始符号。非确定型的自底向上分析器是通过将对下述文法中检查串 $aabb$ 的归约来阐明的

$$G: S \rightarrow aAb \mid BaAa$$

$$A \rightarrow ab \mid b$$

$$B \rightarrow Bb \mid b.$$

595

分析器在找到一个归约子串前会扫描前缀 aab 。 aab 的后缀 b 和 ab 构造出了文法 G 的规则右部。 $aabb$ 的三个归约可以通过移进这些子串来获得。

自底向上分析器的目标是对输入串进行重复归约, 直到获得了开始符号。那么利用规则 $A \rightarrow b$ 进行初始化的 $aabb$ 的一个归约是否最终就会生成开始符号呢? 同样地, $aaAb$ 是 G 的右句型吗? 文法 G 的最右推导具有如下形式:

规则	归约
$A \rightarrow b$	$aaAb$
$A \rightarrow ab$	aAb
$B \rightarrow b$	$aaBb$

$$S \Rightarrow aAb \Rightarrow aabb$$

$$S \Rightarrow aAb \Rightarrow abb$$

$$S \Rightarrow BaAa \Rightarrow Baaba \xRightarrow{i} Bb^i aaba \Rightarrow bb^i aaba \quad i \geq 0$$

$$S \Rightarrow BaAa \Rightarrow Baba \xRightarrow{i} Bb^i aba \Rightarrow bb^i aba \quad i \geq 0.$$

$L(G)$ 中的串的成功推导可以通过在前述推导中“逆向箭头”来实现。因为串 $aaAb$ 和 $aaBb$ 并没有发生在这些推导中, 所以串 $aabb$ 通过使用规则 $A \rightarrow b$ 或者规则 $B \rightarrow b$ 进行初始化的归约是不能够生成 S 的。利用这些附加的信息, 分析器只是需要通过规则 $A \rightarrow ab$ 归约出 aab 即可。

成功的归约是通过检验文法 G 的最右推导来获得的。如果分析器不进行预读, 那么这样的分析器就必须能够确定是否应该在扫描到 uw 的时候就要利用规则 $A \rightarrow w$ 来实现一个归约。下面将要介绍规则 $A \rightarrow w$ 的 LR(0) 上下文集合, 它定义了一种在 w 被扫描器读到的时候就需要执行归约的上下文。

定义 20.1.1 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。如果存在如下的推导:

$$S \xRightarrow{R} uAv \xRightarrow{R} uwv,$$

那么串 uw 是规则 $A \rightarrow w$ 的 LR(0) 上下文。这里 $u \in (V \cup \Sigma)^*$ 并且 $v \in \Sigma^*$ 。规则 $A \rightarrow w$ 的 LR(0) 上下文集合是用 $LR(0) - CONTEXT(A \rightarrow w)$ 来表示的。

规则 $A \rightarrow w$ 的 LR(0) 上下文是通过最右推导来获得的, 最右推导是在应用该规则的时候终止的。对于归约来说, 如果串 uwv 可以归约到 S , 并且这个归约是从利用 A 替代 w 开始的, 那么 uw 是规则

[596]

$A \rightarrow w$ 的 LR(0) 上下文 如果 $uw \notin \text{LR}(0) - \text{CONTEXT}(A \rightarrow w)$, 则不存在这样一个归约序列, 该归约序列以规则 $A \rightarrow w$ 为起点, 并且形如 uwv 的串利用该规则生成 S , 这里 $v \in \Sigma^*$ 。如果知晓了 LR(0) 的上下文, 那么就可以使用该上下文从而利用分析器来消除归约。当 uw 是规则 $A \rightarrow w$ 的 LR(0) 上下文的时候, 分析器只需要使用规则 $A \rightarrow w$ 来归约 uw 。

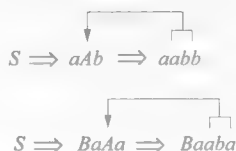
规则 G 的 LR(0) 上下文是由 G 的最右推导构造出来的。为了确定规则 $S \rightarrow aAb$ 的 LR(0) 上下文, 这里考虑所有包含应用规则的最右推导。惟一的两个推导显示如下:

$$S \Rightarrow aAb \Rightarrow aabb$$

$$S \Rightarrow aAb \Rightarrow abb.$$

惟一的终止于规则 $S \rightarrow aAb$ 的最右推导是 $S \Rightarrow aAb$ 。因此, 则 $\text{LR}(0) - \text{CONTEXT}(S \rightarrow aAb) = \{aAb\}$ 成立。

规则 $A \rightarrow ab$ 的 LR(0) 上下文是通过应用规则 $A \rightarrow ab$ 而终止的最右推导来获得的。只有两个这样的推导存在。归约是通过 ab 到 A 的箭头来表示的。上下文是到达此句型的前缀, 并且包括已经归约的 ab 的发生。



因此, 规则 $A \rightarrow ab$ 的 LR(0) 上下文是 aab 和 $Baab$ 。对于 G 中所有的规则, 可以采用相应的方式获得其 LR(0) 上下文。

规则	LR(0) 上下文
$S \rightarrow aAb$	$\{aAb\}$
$S \rightarrow BaAa$	$\{BaAa\}$
$A \rightarrow ab$	$\{aab, Baab\}$
$A \rightarrow b$	$\{ab, Bab\}$
$B \rightarrow Bb$	$\{Bb\}$
$B \rightarrow b$	$\{b\}$

[597]

例 20.1.1 为下述文法的规则构造 LR(0) 上下文

$$S \rightarrow aA \mid bB$$

$$A \rightarrow abA \mid bB$$

$$B \rightarrow bBc \mid bc.$$

利用规则 $S \rightarrow aA$ 进行初始化的最右推导具有如下形式

$$S \Rightarrow aA \Rightarrow a(ab)^i A \Rightarrow a(ab)^i bB \Rightarrow a(ab)^i bb^j Bc^j \Rightarrow a(ab)^i bb^j bcc^j,$$

其中 $i, j \geq 0$ 。从规则 $S \rightarrow bB$ 开始进行的推导如下所示:

$$S \Rightarrow bB \Rightarrow bb^i Bc^i \Rightarrow bb^i bcc^i.$$

LR(0) 的上下文可以从上述推导生成的句型中获得。

规则	LR(0) 上下文
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow abA$	$\{a(ab)^i A \mid i > 0\}$
$A \rightarrow bB$	$\{a(ab)^i bB \mid i \geq 0\}$
$B \rightarrow bBc$	$\{a(ab)^i bb^j Bc, bb^j Bc \mid i \geq 0, j > 0\}$
$B \rightarrow bc$	$\{a(ab)^i bb^j c, bb^j c \mid i \geq 0, j > 0\}$

□

上下文可以用来消除分析器生成的归约。当 LR(0) 上下文提供了足够的信息, 从而可以消除所有的动作, 而只留下一个动作的时候, 这个文法就被称为 LR(0) 文法。

定义 20.1.2 如果对于每一个 $u \in (V \cup \Sigma)^*$, 并且 $v \in \Sigma^*$ 来说,

$u \in \text{LR}(0) - \text{CONTEXT}(A \rightarrow w_1)$, 并且 $uv \in \text{LR}(0) - \text{CONTEXT}(B \rightarrow w_2)$

这也就是说, $v = \lambda$, $A = B$, 并且 $w_1 = w_2$ 。那么具有非递归开始符号 S 的上下文无关文法 $G = (V, \Sigma, P, S)$ 是 LR(0) 的。

例 20.1.1 中的文法是 LR(0) 的。检查 LR(0) 上下文的表, 可以看出不存在某个规则的 LR(0) 上下文是另外某个规则的 LR(0) 的前缀。

LR(0) 文法上下文为选择合适的动作提供了所需的信息。当扫描串 u 的时候, 分析器选择了以下三种彼此独立的动作中的一种:

1. 如果 $u \in \text{LR}(0) - \text{CONTEXT}(A \rightarrow w)$, 那么 u 使用规则 $A \rightarrow w$ 进行归约。
2. 如果 u 不是 LR(0) 上下文, 但却是某个 LR(0) 上下文的前缀, 那么分析器会产生一个移进
3. 如果 u 不是任何 LR(0) 上下文的前缀, 那么输入串将会被拒绝。

因为串 u 是至多一条规则 $A \rightarrow w$ 的 LR(0) 上下文, 所以第一种情况就指明了唯一的动作。如果存在串 $v \in (V \cup \Sigma)^*$, uv 是 LR(0) 上下文的, 那么串 u 被称为活前缀 (viable prefix)。如果 u 是活前缀, 并且不是 LR(0) 上下文的, 那么一个移进操作序列可以生成 LR(0) 上下文 uv 。

例 20.1.2 文法

$G: S \rightarrow aA \mid aB$

$A \rightarrow aAb \mid b$

$B \rightarrow bBa \mid b$

不是 LR(0) 的。对于任意 $i \geq 0$, G 的最右推导具有如下形式:

$S \Rightarrow aA \xRightarrow{i} aa^i Ab^i \Rightarrow aa^i bb^i$

$S \Rightarrow aB \xRightarrow{i} ab^i Ba^i \Rightarrow ab^i ba^i$

该文法规则的 LR(0) 上下文可以通过上述推导的右句型获得

文法 G 不是 LR(0) 的, 这是因为 ab 是一个包含规则 $B \rightarrow b$ 并且包含规则 $A \rightarrow b$ 的 LR(0) 上下文。□

规则	LR(0) 上下文
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow aB$	$\{aB\}$
$A \rightarrow aAb$	$\{aa^i Ab \mid i > 0\}$
$A \rightarrow b$	$\{aa^i b \mid i \geq 0\}$
$B \rightarrow bBa$	$\{ab^i Ba \mid i > 0\}$
$B \rightarrow b$	$\{ab^i \mid i > 0\}$

20.2 LR(0) 分析器

将 LR(0) 文法规则的 LR(0) 上下文提供的信息合成到底自底向上分析器中将会产生确定型的分析算法。输入串 p 是按照从左到右的方式进行扫描的。算法 20.2.1 中的分析器的动作是通过将 LR(0) 上下文和扫描的串相比较来确定的。串 u 是被分析器扫描的句型的前缀, v 是输入串的余下部分。操作 $\text{shift}(u, v)$ 将第一个符号从 v 中移除, 并将它连接到 u 的右端。

算法 20.2.1

LR(0) 文法的分析器

输入: LR(0) 文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

1. $u := \lambda$, $v := p$
2. $\text{dead} - \text{end} := \text{false}$
3. **repeat**
 - 3.1. **if** 对于 P 中的规则 $A \rightarrow w$ 有 $u \in \text{LR}(0) - \text{CONTEXT}(A \rightarrow w)$
 其中 $u = xw$ **then** $u := xA$
else if u 是变量前缀且 $v \neq \lambda$ 那么 $\text{shift}(u, v)$
else $\text{dead} - \text{end} := \text{true}$
- until** $u = S$ or $\text{dead} - \text{end}$
4. **if** $u = S$ **then** 接收 **else** 拒绝

如果遇到子串 $u = xw$, 就要利用规则 $A \rightarrow w$ 进行归约。做出此决定不需要使用 v 中包含的任何信息,

598

599

v 是串中未被扫描的部分。分析器并不预先读取串 av ，这样 LR(0) 中的零就表示不需要进行预先读取。

算法 20.2.1 忽略了一个细节。该算法没有提供可以确定一个串是活前缀还是文法规则的 LR(0) 上下文的技术。在下一节中，我们将会设计一种有限自动机，该有限自动机的计算可以确定 LR(0) 的上下文和活前缀。

例 20.2.1 串 $aabbbbcc$ 是通过使用例 20.1.1 中的文法的规则和 LR(0) 上下文以及用于 LR(0) 文法的算法来进行分析的。

u	v	规则	行为
λ	$aabbbbcc$		转移
a	$abbbbcc$		转移
aa	$bbbbcc$		转移
aab	$bbbcc$		转移
$aabb$	$bbcc$		转移
$aabbb$	bcc		转移
$aabbbb$	cc		转移
$aabbbbc$	c	$B \rightarrow bc$	归约
$aabbbbB$	c		转移
$aabbbbBc$	λ	$B \rightarrow bBc$	归约
$aabbbB$	λ	$A \rightarrow bB$	归约
$aabA$	λ	$A \rightarrow abA$	归约
aA	λ	$S \rightarrow aA$	归约
S			

20.3 LR(0) 机

为了选择合适的动作，LR(0) 分析器将处理的串 u 与文法规则的 LR(0) 上下文相比较。因为规则的 LR(0) 上下文集合可以包含无穷多的串，而且集合中的串可以任意长，这里不能生成这些集合从而直接进行比较。处理无穷集合的问题可以在 LR(k) 文法中得以避免，这是通过限制预读串长度的方式来实现的。令人遗憾的是，对串进行归约的决定要求知晓整个串（上下文）的扫描过程。LR(0) 文法 G_1 和 G_2 可以演示这种依赖关系。

文法 G_1 的规则 $A \rightarrow aAb$ 和规则 $A \rightarrow ab$ 的 LR(0) 上下文形成了不相交集，这些不相交集可以满足前缀的情况。如果这些集合被截成任意长度 k ，那么串 a^k 将会是这两个截取集合中的元素。上下文的最后两个符号需要用来区别这些归约。

可能的读者只是考虑了上下文的定长后缀的情况，这是因为归约会更改扫描串的后缀。文法 G_2 可以显示出这种方法的不可用性。

规则 $A \rightarrow ab$ 和规则 $B \rightarrow ab$ 的 LR(0) 上下文的唯一不同就是串中的第一个元素。如果选择的过程只是使用 LR(0) 上下文的定长中缀，那么分析器将不能区别这些规则。

文法 G_1 和 G_2 演示了 LR(0) 分析器需要整个扫描串来选择合适的动作。令人遗憾的是，这并不是说需要整个 LR(0) 上下文集合。对于一个给定的文法，可以构造一个有限自动机，该有限自动机的计算可以确定某个串是否是文法的活前缀。该机的状态被称为 LR(0) 项，可以直接从

规则	LR(0) 上下文
$G_1: S \rightarrow A$	$\{A\}$
$A \rightarrow aAa$	$\{a^i A a \mid i > 0\}$
$A \rightarrow aAb$	$\{a^i A b \mid i > 0\}$
$A \rightarrow ab$	$\{a^i b \mid i > 0\}$

规则	LR(0) 上下文
$G_2: S \rightarrow A$	$\{A\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow aA$	$\{a^i A \mid i > 0\}$
$A \rightarrow ab$	$\{a^i b \mid i > 0\}$
$B \rightarrow aB$	$\{ba^i B \mid i > 0\}$
$B \rightarrow ab$	$\{ba^i b \mid i > 0\}$

600

601

文法规则构造该机。

定义 20.3.1 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。 G 的 LR(0) 项可以进行如下定义:

- i) 如果 $A \rightarrow uv \in P$, 那么 $A \rightarrow u \cdot v$ 是 LR(0) 项。
- ii) 如果 $A \rightarrow \lambda \in P$, 那么 $A \rightarrow \cdot$ 是 LR(0) 项。

LR(0) 项可以从文法的规则中获得, 这是通过在规则的右边放置一个标记 “ \cdot ” 来实现的。一项 “ $A \rightarrow u \cdot$ ” 被称为完全项 (complete item), 对于一个右部长度为 n 的规则可以生成 $n+1$ 个项, 从而用于标记的每一个可能的位置。

定义 20.3.2 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。 G 的非确定型 LR(0) 机 (nondeterministic LR(0) machine) 是 $NFA - \lambda M = (Q, V \cup \Sigma, \delta, q_0, Q)$, 这里 Q 是从 q_0 开始增大的 LR(0) 项集合。状态转换函数定义如下:

- i) $\delta(q_0, \lambda) = \{S \rightarrow \cdot w \mid S \rightarrow w \in P\}$
- ii) $\delta(A \rightarrow u \cdot av, a) = \{A \rightarrow ua \cdot v\}$
- iii) $\delta(A \rightarrow u \cdot Bv, B) = \{A \rightarrow uB \cdot v\}$
- iv) $\delta(A \rightarrow u \cdot Bv, \lambda) = \{B \rightarrow \cdot w \mid B \rightarrow w \in P\}$ 。

对于任意文法的活前缀的串来说, 文法 G 的非确定型 LR(0) 机 M 的计算完全可以处理这样的串。所有其他的计算在读入整个输入之前就停止了。因为 M 的所有状态都被接收, 所以 M 正好接收以前文法的所有活前缀。 M 的计算记录了对匹配文法 G 的规则右部而做出了进展。一项 $A \rightarrow u \cdot v$ 则表示了串 u 已经被扫描了, 并且自动机正在寻找串 v 来完成匹配。

在这里, 跟在标记之后的符号定义了从一个节点出来的弧。如果标记在终结符之前, 那么从节点出来的唯一的弧就用那个终结符来标记。标记的弧 B 或者 λ 可能在从节点出来的时候包含了具有形式 $A \rightarrow u \cdot Bv$ 的项。为了扩展规则的右部匹配, 该机会寻找一个 B 。如果分析器读到了 B , 那么节点 $A \rightarrow uB \cdot v$ 就被输入进来。除此之外, 还要寻找可以生成 B 的串。变量 B 可以通过使用一个规则 B 的归约来获得。这样, 分析器也寻找规则 B 的右部。这是通过对项 $B \rightarrow \cdot w$ 进行 λ 状态转换来实现的。

在定义 20.3.2 中, 下表中给出的 LR(0) 项, 以及文法 G 的规则 LR(0) 上下文, 从而来演示活前缀的识别, 而这是通过使用相应的 $NFA - \lambda$ 来实现的。

规则	LR(0) 项	LR(0) 上下文
$S \rightarrow AB$	$S \rightarrow \cdot AB$	$\{AB\}$
	$S \rightarrow A \cdot B$	
	$S \rightarrow AB \cdot$	
	$S \rightarrow \cdot$	
$A \rightarrow Aa$	$A \rightarrow \cdot Aa$	$\{Aa\}$
	$A \rightarrow A \cdot a$	
	$A \rightarrow Aa \cdot$	
	$A \rightarrow \cdot$	
$A \rightarrow a$	$A \rightarrow \cdot a$	$\{a\}$
	$A \rightarrow a \cdot$	
	$A \rightarrow \cdot$	
	$A \rightarrow \cdot$	
$B \rightarrow bBa$	$B \rightarrow \cdot bBa$	$\{Ab'Ba \mid i > 0\}$
	$B \rightarrow b \cdot Ba$	
	$B \rightarrow bB \cdot a$	
	$B \rightarrow bBa \cdot$	
	$B \rightarrow \cdot$	
$B \rightarrow ba$	$B \rightarrow \cdot ba$	$\{Ab'ba \mid i \geq 0\}$
	$B \rightarrow b \cdot a$	
	$B \rightarrow ba \cdot$	
	$B \rightarrow \cdot$	

图 20-1 中给出的 $NFA - \lambda$ 是文法 G 的 LR(0) 机。如果 $A \rightarrow u \cdot v \in \delta(q, w)$, 那么串 w 是规则 $A \rightarrow uv$ 的上下文的前缀。图 20-1 中的 LR(0) 机的计算在遇到包含项 $A \rightarrow \cdot a$, $S \rightarrow A \cdot B$, $B \rightarrow \cdot bBa$ 和 $B \rightarrow \cdot ba$ 的时候停机。这些正好是具有从 A 开始的 LR(0) 上下文的规则。同样地, 具有输入 AbB 的计算显示了

AbB 是规则 $B \rightarrow bBa$ 的活前缀, 除此之外, 再无其他。

第 5 章中介绍的技术可以用于构造 G 的非确定型 LR(0) 机的一个等价的 DFA。图 20-2 描述了该机, 即 G 的确定型 LR(0) 机 (deterministic LR(0) machine)。确定型机的开始状态 q_s 是 q_0 的 λ -闭包, q_0 是非确定型机的初态。空集代表失败状态, 在这里被省略。当处理串 u 的计算成功终止的时候, u 即是 LR(0) 上下文或者活前缀。算法 20.3.3 将 LR(0) 机的信息合成进了 LR(0) 分析策略中。

603

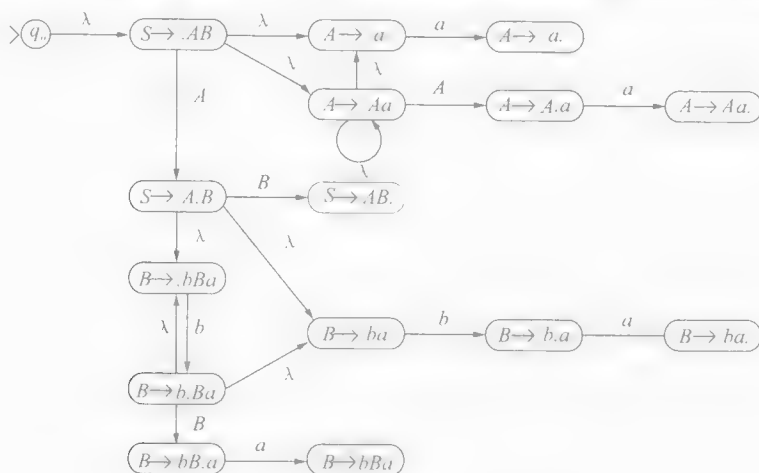


图 20-1 G 的非确定型 LR(0) 机

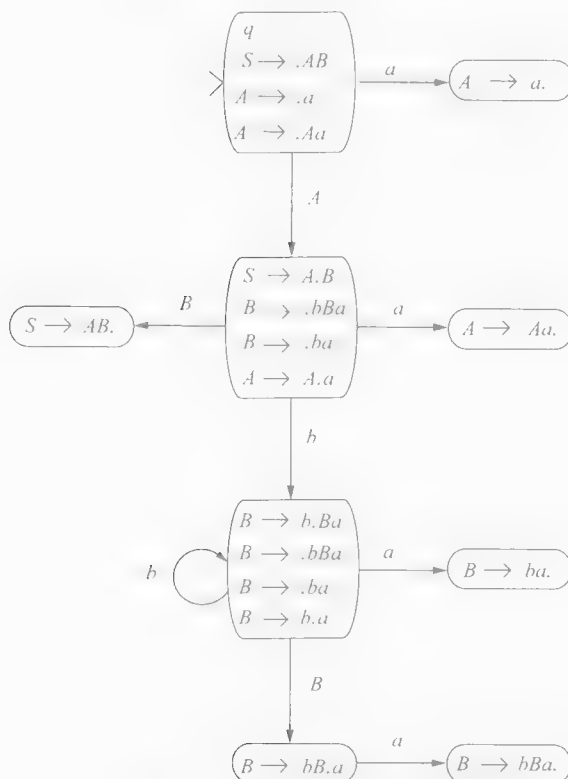


图 20-2 G 的确定型 LR(0) 机

算法 20.3.3

使用确定型 LR(0) 机的分析器

输入: LR(0)文法 $G = (V, \Sigma, P, S)$ 字符串 $p \in \Sigma^*$ G 的确定型 LR(0) 机器1. $u := \lambda, v := p$ 2. $\text{dead-end} := \text{false}$

3. repeat

3.1. if $\hat{\delta}(q_s, u)$ 包含 $A \rightarrow w$, 其中 $u = xw$ then $u := xA$ else if $\hat{\delta}(q_s, u)$ 包含一个项 $A \rightarrow y.z$, 且 $v \neq \lambda$ then 转移 (u, v) else $\text{dead-end} := \text{true}$ until $u = S$ or dead-end 4. if $u = S$ then 接收 else 拒绝

604

步骤 3.1 用于确定要采用哪种动作, 这是基于 LR(0) 机计算 $\hat{\delta}(q_s, u)$ 的结果。如果 $\delta(q_s, u)$ 包含了一个完全 LR(0) 项 $A \rightarrow w$, 那么就要进行使用规则 $A \rightarrow w$ 进行的归约, 循环就重复进行以得到结果串。如果 $\delta(q_s, u)$ 包含了 LR(0) 项 $A \rightarrow y.z$, 那么就要执行一个转换函数来扩展活前缀的匹配。最后, 如果 $\hat{\delta}(q_s, u)$ 为空, 那么计算停止。

例 20.3.1 串 $uabbbaa$ 使用算法 20.3.3 和图 20-2 中的确定型的 LR(0) 机来分析。当处理第一个 a 的时候, 该机进入状态 $A \rightarrow a$, 使用规则 $A \rightarrow a$ 来指定归约。因为 $\hat{\delta}(q_s, A)$ 不包含完全项, 所以分析器进行移进并且构造串 Aa 。计算 $\delta(q_s, Aa) = \{A \rightarrow Aa\}$ 表示了 Aa 是 $A \rightarrow Aa$ 的 LR(0) 上下文, 并且不是其他任何规则的上下文的前缀。在生成了一个完全项以后, 分析器使用规则 $A \rightarrow Aa$ 来归约串。移进和归约循环持续进行, 直到句型已经被归约为开始符号 S 。□

605

u	v	计算	行为
λ	$aabbbaa$	$\hat{\delta}(q_s, \lambda) =$	$\{S \rightarrow AB,$ $A \rightarrow a,$ $A \rightarrow Aa\}$ 移动
a	$abbbaa$	$\hat{\delta}(q_s, a) =$	$\{A \rightarrow a\}$ 归约
A	$abbbaa$	$\hat{\delta}(q_s, A) =$	$\{A \rightarrow A.a,$ $S \rightarrow A.B,$ $B \rightarrow bBa,$ $B \rightarrow ba\}$ 移动
Aa	$bbbaa$	$\hat{\delta}(q_s, Aa) =$	$\{A \rightarrow Aa\}$ 归约
A	$bbbaa$	$\hat{\delta}(q_s, A) =$	$\{A \rightarrow A.a,$ $S \rightarrow A.B,$ $B \rightarrow bBa,$ $B \rightarrow ba\}$ 移动
Ab	baa	$\hat{\delta}(q_s, Ab) =$	$\{B \rightarrow bBa,$ $B \rightarrow b.Ba,$ $B \rightarrow ba,$ $B \rightarrow b.a\}$ 移动
Abb	aa	$\hat{\delta}(q_s, Abb) =$	$\{B \rightarrow bBa,$ $B \rightarrow b.Ba,$ $B \rightarrow ba,$ 移动

(续)

u	v	计算	行为
			$B \rightarrow b.a$
$Abba$	a	$\hat{\delta}(q_s, Abba) =$	$\{B \rightarrow ba.\}$ 归约
AbB	a	$\hat{\delta}(q_s, AbB) =$	$\{B \rightarrow bB.a\}$ 移动
$AbBa$	λ	$\hat{\delta}(q_s, AbBa) =$	$\{B \rightarrow bBa.\}$ 归约
AB	λ	$\hat{\delta}(q_s, AB) =$	$\{S \rightarrow AB.\}$ 归约
S			

[7]

20.4 被 LR(0) 机接收

[606]

构造 LR(0) 机用于确定串是否是文法的活前缀。定理 20.4.1 建立了 LR(0) 机的计算, 用以提供所需要的信息。

定理 20.4.1 设 G 是上下文无关文法, M 是 G 的非确定型 LR(0) 机。LR(0) 项 $A \rightarrow u.v$ 在 $\hat{\delta}(q_i, w)$ 中, 当且仅当 $w = pu$, 这里 puv 是 $A \rightarrow uv$ 的 LR(0) 上下文。

证明: 设 $A \rightarrow u.v$ 是 $\hat{\delta}(q_0, w)$ 的元素。通过对计算 $\hat{\delta}(q_0, w)$ 中的状态转换数进行归纳证明, 可以得出 wv 是 $A \rightarrow uv$ 的 LR(0) 上下文。

归纳基础包含了计算长度为一的情况。所有这样的计算都具有如下形式:

其中 $S \rightarrow q$ 是文法的一个规则。这些计算处理输入串 $w = \lambda$ 。设置 $p = \lambda$ 、 $u = \lambda$ 和 $v = q$ 给出了 w 的分解。



现在, 设 $\hat{\delta}(q_0, w)$ 是 $\hat{\delta}(q_i, w)$ 中带有 $A \rightarrow u.v$ 且长度 $k > 1$ 的计算。将最后一个转换隔离开, 我们可以将计算写成 $\hat{\delta}(\hat{\delta}(q_0, y), \lambda)$, 其中 $w = yx$, 并且 $x \in V \cup \Sigma \cup \{\lambda\}$ 。证明的余下部分被分成下面几种情况。

情况 1: $x \in V$ 。在这种情况下, $u = u'a$ 。计算的最后转换具有如下形式:

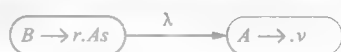
根据归纳假设, $pu'av = wv$ 是 $A \rightarrow uv$ 的 LR(0) 上下文。



情况 2: 证明类似情况 1。

情况 3: $x = \lambda$ 。如果 $x = \lambda$, 那么 $y = w$, 并且计算在项 $B \rightarrow rA$ 处终止。最终转换具有如下形式: 归纳假设暗示了 w 可以写成 $w = pr$, 这里 $prAs$ 是 $B \rightarrow rAs$ 的 LR(0)

上下文。这样, 就存在一个最右推导, 形式如下所示:



$$S \xRightarrow{R} pBq \xRightarrow{R} prAsq.$$

对规则 $A \rightarrow v$ 的应用生成了

$$S \xRightarrow{R} pBq \xRightarrow{R} prAsq \xRightarrow{R} prvsq.$$

这个推导的最终步骤表明了 $prv = wv$ 是 $A \rightarrow v$ 的 LR(0) 上下文。

为了确立对等的含义, 必须要证明不管何时, 只要 puv 是规则 $A \rightarrow uv$ 的 LR(0) 上下文, 那么 $\hat{\delta}(q_0, pu)$ 就包含项 $A \rightarrow u.v$ 。首先, 需要标记出, 如果 $\hat{\delta}(q_0, p)$ 包含了 $A \rightarrow .uv$, 那么 $\hat{\delta}(q_0, w)$ 包含了规则 $A \rightarrow .uv$ 。这是紧随在定义 20.3.2 中的情况 (ii) 和 (iii) 之后的。

[607]

因为 puv 是 $A \rightarrow uv$ 的 LR(0) 上下文, 那么存在一个如下所示的推导

$$S \xRightarrow{R} pAq \xRightarrow{R} puvq.$$

通过对推导 $S \xRightarrow{R} pAq$ 的长度进行归纳, 可以证明 $\hat{\delta}(q_0, p)$ 包含 $A \rightarrow .uv$ 。归纳基础对长度为一的情况进行推导, 即 $S \xRightarrow{R} pAq$ 。所要求的计算需要能够遍历处理串 p 的过程中, 能够到达 $S \xRightarrow{R} pAq$ 的那些弧。计算是通过跟踪从 $S \xRightarrow{R} p.Aq$ 到 $A \rightarrow .uv$ 的 λ -弧来完成的。

现在, 考虑变量 A 出现在第 k 个规则上的推导。这种形式的推导可以写成如下形式

$$S \xRightarrow{R} xBy \xRightarrow{R} xwAzy.$$

这种归纳假设声明了 $\hat{\delta}(q_0, x)$ 包含了项 $B \rightarrow w.Az$ 。因此 $B \rightarrow w.Az \in \hat{\delta}(q_0, xw)$ 。 λ -状态转换到 $A \rightarrow .uv$

以完成这个计算。 ■

在引理 20.4.2 中, 上下文无关文法中的推导和文法的确定型 LR(0) 机中的节点的项之间的关系紧随引理 20.4.1。引理 20.4.2 的证明留做练习。 q_0 是确定型机的开始符号。

引理 20.4.2 设 M 是上下文无关文法 G 的确定型 LR(0) 机。假设 $\delta(q_0, w)$ 包含了项 $A \rightarrow u.Bv$

i) 如果 $B \Rightarrow \lambda$, 那么对于某些变量 $C \in V$ 来说, $\hat{\delta}(q_s, w)$ 包含了具有形式 $C \rightarrow \cdot$ 的项。

ii) 如果 $B \rightarrow x \in \Sigma^+$, 那么对于某些变量 $C \in V$ 来说, 存在一个被终结符标记的弧, 这些弧从节点 $\hat{\delta}(q_s, w)$ 出来, 或者 $\hat{\delta}(q_s, w)$ 包含了具有形式 $C \rightarrow \cdot$ 的项。

引理 20.4.3 设 M 是 LR(0) 文法 G 的确定型 LR(0) 机。假设 $\delta(q_0, u)$ 包含了完全项 $A \rightarrow w$ 。那么 $\hat{\delta}(q_s, ua)$ 对于所有的终结符 $a \in \Sigma$ 来说都是未定义的。

证明: 根据引理 20.4.1, u 是 $A \rightarrow w$ 的 LR(0) 上下文。假设 $\delta(q_0, ua)$ 用于定义某个终结符 a 。那么 ua 就是某个规则 $B \rightarrow y$ 的 LR(0) 上下文的前缀。这就意味着存在如下的推导:

$$S \xRightarrow{R} pBv \Rightarrow pyv \Rightarrow uazv$$

这里 $z \in (V \cup \Sigma)^*$, 并且 $v \in \Sigma^+$ 。考虑到对于串 z 的可能性, 如果 $z \in \Sigma^+$, 那么 uaz 是规则 $B \rightarrow y$ 的 LR(0) 上下文。如果 z 不是终结符串, 那么存在一个从 z 推导出来的终结符串

$$z \xRightarrow{R} rCs \rightarrow rts \quad r, s, t \in \Sigma^+,$$

[608]

这里 $C \rightarrow t$ 是从 z 推导终结符串的过程中最后应用的规则。将 S 的推导和 z 联合起来, 可以看出 $uazrt$ 是 Σ^+ 的 LR(0) 上下文。无论哪种情况, u 都是 LR(0) 上下文, 并且 ua 是活前缀。这与假设 G 是 LR(0) 的相矛盾。 ■

前面的结果可以与定义 20.1.2 结合起来, 以便于获得 LR(0) 文法在确定型 LR(0) 机的结构方面的描述。

定理 20.4.4 设 G 是具有非递归开始符号的上下文无关文法。 G 是 LR(0) 文法的, 当且仅当 G 的确定型 LR(0) 机的扩展的状态转换函数 δ 满足下列情况:

i) 如果 $\hat{\delta}(q_s, u)$ 包含完全项 $A \rightarrow w$, 并且 $w \neq \lambda$, 那么 $\hat{\delta}(q_s, u)$ 不包含其他项。

ii) 如果 $\hat{\delta}(q_s, u)$ 包含完全项 $A \rightarrow \cdot$, 那么 $\hat{\delta}(q_s, u)$ 中所有其他项的标记后面都要跟上一个变量。

证明: 首先, 要证明当扩展的状态转换函数满足情况 (i) 和 (ii) 的时候, 具有非递归开始符号的文法 G 是 LR(0) 的。设 u 是规则 $A \rightarrow w$ 的 LR(0) 上下文。那么 $\delta(q_0, uv)$ 只有在 v 以变量开头的时候才被定义。这样, 对于所有的串 $v \in \Sigma^+$, $uv \in \text{LR}(0) - \text{CONTEXT}(B \rightarrow x)$ 就意味着 $v = \lambda$, $B = A$ 并且 $w = x$ 。

反过来, 设 G 是 LR(0) 文法, 并且 u 是规则 $A \rightarrow w$ 的 LR(0) 上下文。根据定理 20.4.1, $\delta(q_0, u)$ 包含了完全项 $A \rightarrow w$ 。状态 $\hat{\delta}(q_s, u)$ 不包含任何其他完全项 $B \rightarrow v$ 。因为这意味着 u 也是 $B \rightarrow v$ 的 LR(0) 上下文。根据引理 20.4.3, 所有从 $\delta(q_0, u)$ 出来的弧必须使用变量来标记。

现在, 假设 $\hat{\delta}(q_s, u)$ 包含完全项 $A \rightarrow w$, 其中 $w \neq \lambda$ 。根据引理 20.4.2, 如果存在一个弧, 它被以 $\hat{\delta}(q_s, u)$ 结尾的变量标记, 那么 $\hat{\delta}(q_s, u)$ 包含了完全项 $C \rightarrow \cdot$ 或者 $\delta(q_s, u)$ 具有一个从它出来, 并且被终结符标记的弧。在以前的情况中, u 即是 $A \rightarrow w$ 的 LR(0) 上下文, 也是 $C \rightarrow \lambda$ 的 LR(0) 上下文, 这与假设 G 是 LR(0) 的相矛盾。后面一种情况与引理 20.4.3 相矛盾。这样, $A \rightarrow w$ 是 $\delta(q_s, u)$ 中的惟一一项。 ■

直观上来看, 可以这样说, 如果一个包含一个完全项的状态不包含其他项, 那么该文法是 LR(0) 的。对于由非空规则生成的完全项来说, 这种情况被所有包含该完全项的状态满足。对于包含 $A \rightarrow \cdot$ 的状态来说, 前面的定理允许该状态包含标记后跟随变量的项。考虑使用规则 $S \rightarrow aABc$, $A \rightarrow \lambda$ 和 $B \rightarrow b$ 的推导, 显示如下:

$$S \xRightarrow{R} aABc \xRightarrow{R} aAbc \xRightarrow{R} abc$$

串 a 是 $A \rightarrow \lambda$ 的 LR(0) 上下文, 并且是 aAb 的前缀, 而 aAb 是 $B \rightarrow b$ 的 LR(0) 上下文。例 20.4.1 可以

609 演示使用 LR(0) 分析器中的 λ -规则进行归约的效果。

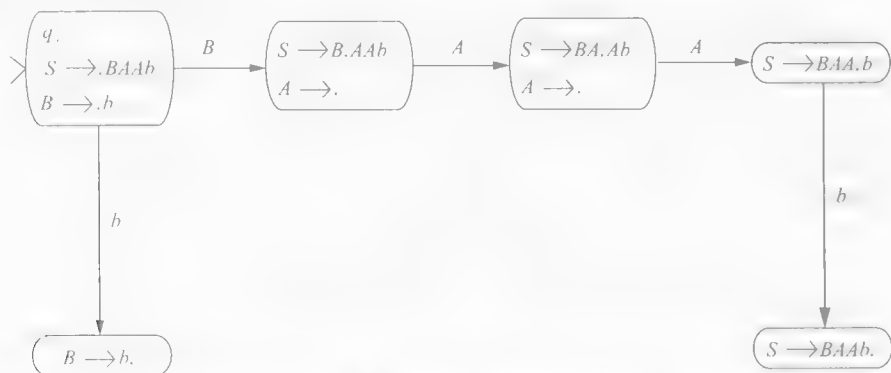
例 20.4.1 给出如下文法：

$G: S \rightarrow BAAb$

$A \rightarrow \lambda$

$B \rightarrow b$

其确定型的 LR(0) 机显示如下：



利用该机的计算来跟踪串 bb 的分析，该分析可以用于详细说明分析器的动作。

u	v	计算	行为
λ	bb	$\hat{\delta}(q_s, \lambda) =$	$\{S \rightarrow \cdot BAAb$ $B \rightarrow \cdot b\}$ 移动
b	b	$\hat{\delta}(q_s, b) =$	$\{B \rightarrow \cdot b.\}$ 归约
B	b	$\hat{\delta}(q_s, B) =$	$\{S \rightarrow B \cdot AAb$ $A \rightarrow \cdot\}$ 归约
BA	b	$\hat{\delta}(q_s, BA) =$	$\{S \rightarrow BA \cdot Ab$ $A \rightarrow \cdot\}$ 归约
BAA	b	$\hat{\delta}(q_s, BAA) =$	$\{S \rightarrow BAA \cdot b\}$ 移动
$BAAb$	λ	$\hat{\delta}(q_s, BAAb) =$	$\{S \rightarrow BAAb \cdot\}$ 归约
S			

每当 LR(0) 机停机于一个包含完全项 $A \rightarrow \lambda$ 的状态的时候，分析器均使用规则 $A \rightarrow \cdot$ 归约句型。此归约将 A 加入到当前扫描串的尾部。在下一个迭代中，LR(0) 机跟踪用 A 标记的弧，从而到达后来的状态。只有当 A 的出现增加了可以识别的项的前缀的时候， A 才由 λ 归约生成。□

定理 20.4.4 确立了一个用于确定文法是否是 LR(0) 的程序。这个程序从构造文法的确定型 LR(0) 机开始。如果定理 20.4.4 中的情况 (i) 和 (ii) 带来的限制被 LR(0) 机满足，那么具有非递归开始符号的文法是 LR(0) 的。

例 20.4.2 利用标记 # 扩展的文法 AE

$AE: S \rightarrow A\#$

$A \rightarrow A + T \mid T$

$T \rightarrow b \mid (A),$

是 LR(0) 的。AE 的确定型 LR(0) 机在图 20-3 中给出。因为每个包含完全项的状态是一个独集合，文法是 LR(0) 的。□

例 20.4.3 文法

$S \rightarrow A\#$

$A \rightarrow A + T \mid T$

$T \rightarrow T \cdot F \mid F$

20.5 LR(1)文法

LR(0)的情况对于构造用来定义程序编程语言的文法来说,限制太多。在本节中,我们对 LR 分析器做了修改,从而可以利用预读子串所获得的信息,这个子串可以匹配规则的右部。预读受限于单符号。具有明显更改的定义和算法可以扩展从而利用任何长度的预读集合。

对于一个文法,如果该文法中的串可以使用一个符号长度的预读来进行确定型分析,那么这个文法就被称为 LR(1)的。预读符号是紧靠将要被分析器归约的子串的右边的符号。当扫描一个具有形式 $uvwz$ 的串的时候,就要使用规则 $A \rightarrow w$ 进行归约,其中, $z \in \Sigma \cup \{\lambda\}$ 。在理解了 LR(0)文法的例子之后,如果存在如下这样一个推导

$$S \xRightarrow{R} uAv \xRightarrow{R} uwv,$$

那么串 $uvwz$ 被称为 LR(1)上下文 (LR(1) context), 其中 z 是 v 的第一个符号, 如果 $v = \lambda$, 那么 z 就是空串。因为自底向上分析器构造的推导是最右推导, 所以预读符号 z 要么是一个终结符符号, 要么是空串。

预读符号的作用就是用于减少分析器必须要检查的可能性的数目, 这是通过在下述文法中考虑归约来证明的:

$$\begin{aligned} G: S &\rightarrow A \mid Bc \\ A &\rightarrow aA \mid a \\ B &\rightarrow a \mid ab. \end{aligned}$$

当 LR(0)分析器读取符号 a , 此时, 存在三种可能情况:

- i) 使用 $A \rightarrow a$ 进行归约
- ii) 使用 $B \rightarrow a$ 进行归约
- iii) 移进以获得 aA 或者 ab

一个符号的预读对于确定合适的操作来说是足够的。当初始 a 被分析器扫描到时, 那么在下面每一个推导中, 加下划线的符号就是预读符号。

$$\begin{array}{llll} S \Rightarrow A & S \Rightarrow A & S \Rightarrow Bc & S \Rightarrow Bc \\ \Rightarrow a_ & \Rightarrow aA & \Rightarrow a_c & \Rightarrow a_bc \\ & \Rightarrow aaA & & \\ & \Rightarrow aqa & & \end{array}$$

在前面的语法中, 当读取到一个 a 的时候, 分析器的动作完全是由预读符号来决定的

LR(0)分析器的动作是由文法的 LR(0) 机的计算结果决定的。LR(1)分析器将预读符号合成到确定程序中。LR(1)项是一个有序对, 它包含了 LR(0)项和一个包含可能的预读符号的集合。

定义 20.5.1 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法。G 的 LR(1)项 (LR(1) Item) 具有如下形式

$$[A \rightarrow u.v, \{z_1, z_2, \dots, z_n\}],$$

其中, $A \rightarrow uv \in P$, 并且 $z_i \in \Sigma \cup \{\lambda\}$ 。集合 $\{z_1, z_2, \dots, z_n\}$ 是 LR(1)集合的预读集合。

项 $[A \rightarrow u.v, \{z_1, z_2, \dots, z_n\}]$ 的预读集合包括最右推导中跟随 uv 的终结符串 y 的第一个符号。

$$S \xRightarrow{R} xAy \xRightarrow{R} xuvy$$

因为 S 规则是非递归的, 所以惟一被规则 $S \rightarrow w$ 终止的推导是 $S \Rightarrow w$ 。该推导中空串跟随在 w 之后。这样, 规则 S 的预读集合一直都是独集合 $\{\lambda\}$ 。

像以前一样, 在项里面的完全项中, 标记跟随在规则的整个右部。LR(1)机指明了 LR(1)分析器

的动作,它是由文法的 LR(1)项构造的。

定义 20.5.2 设 $G = (V, \Sigma, P, S)$ 是上下文无关文法 G 的非确定型 LR(1)机是 $NFA - \lambda M = (Q, V \cup \Sigma, \delta, q_0, Q)$, 这里 Q 是利用状态 q_0 增长的 LR(1)项的集合。状态转换函数定义如下

- i) $\delta(q_0, \lambda) = \{ [S \rightarrow \cdot w, \{ \lambda \}] \mid S \rightarrow w \in P \}$
- ii) $\delta([A \rightarrow u \cdot Bv, \{ z_1, \dots, z_n \}], B) = \{ [A \rightarrow uB \cdot v, \{ z_1, \dots, z_n \}] \}$
- iii) $\delta([A \rightarrow u \cdot av, \{ z_1, \dots, z_n \}], a) = \{ [A \rightarrow ua \cdot v, \{ z_1, \dots, z_n \}] \}$
- iv) $\delta([A \rightarrow u \cdot Bv, \{ z_1, \dots, z_n \}], \lambda) = \{ [B \rightarrow \cdot w, \{ y_1, \dots, y_k \}] \mid B \rightarrow w \in P$, 这里, 对于某些 j 来说 $y_j \in \text{FIRST}_1(vz_j)$

如果忽视预读集合, 那么 (i)、(ii) 和 (iii) 中定义的 LR(1)机的状态转换具有和 LR(0)机相同的形式 LR(1)项 $A \rightarrow u \cdot v, z_1, z_2, \dots, z_n$ 表明了分析器已经扫描过串 u , 并且正在尝试要找到 v 从而完成规则的右部匹配 情况 (ii) 和 (iii) 生成的状态转换表现了匹配规则的右部所进行的中间步骤, 并且没有更改预读集合。情况 (iv) 引入了如下形式的状态转换:



紧跟在这个弧之后, LR(1)机跟随此弧, 并且尝试匹配规则 $B \rightarrow w$ 的右部。如果串 w 找到了, 那么归约 uvw 则按要求生成了 $uB \cdot v$ 预读集合包含了跟随 w 的符号, 也就是说, 如果 $u \Rightarrow v$, 那么串中第一个终结符是由 v 和预读集合 $\{ z_1, \dots, z_n \}$ 推导出来的。

每当 $A \rightarrow w$ 是文法的规则, 自底向上分析器就可能将串 uw 归约到 uA 当发生这种情况的时候, LR(1)分析器使用预读集合来决定是否要归约还是要转换 如果 $\delta(q_i, uw)$ 包含了完全项 $[A \rightarrow w \cdot, \{ z_1, z_2, \dots, z_n \}]$, 则串被归约当且仅当预读符号在集合 $\{ z_1, \dots, z_n \}$ 中。

文法 G 的非确定型和确定型 LR(1)机的状态转换图如图 20-4 和图 20-5 所示

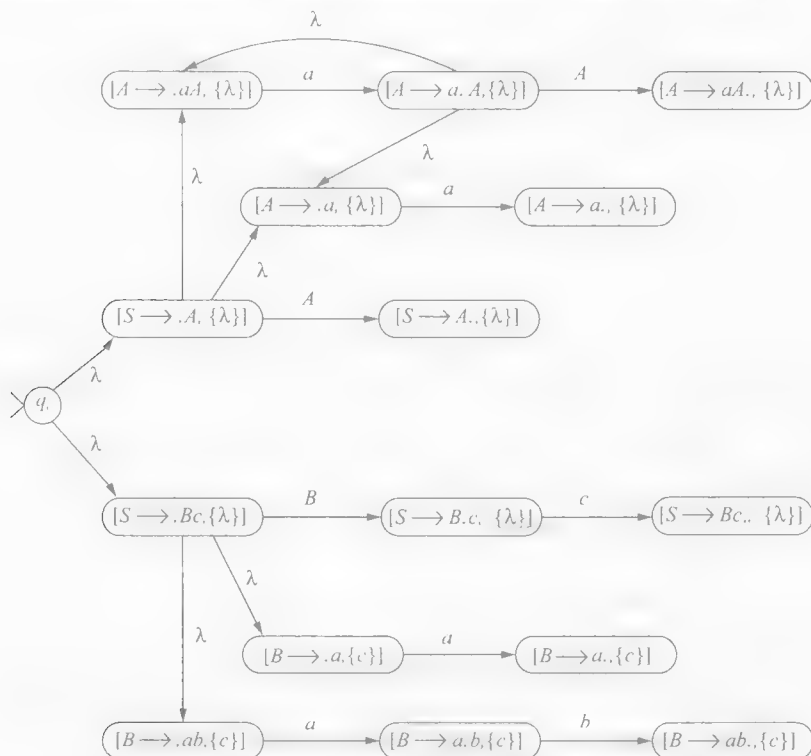


图 20-4 G 的非确定型 LR(1)机

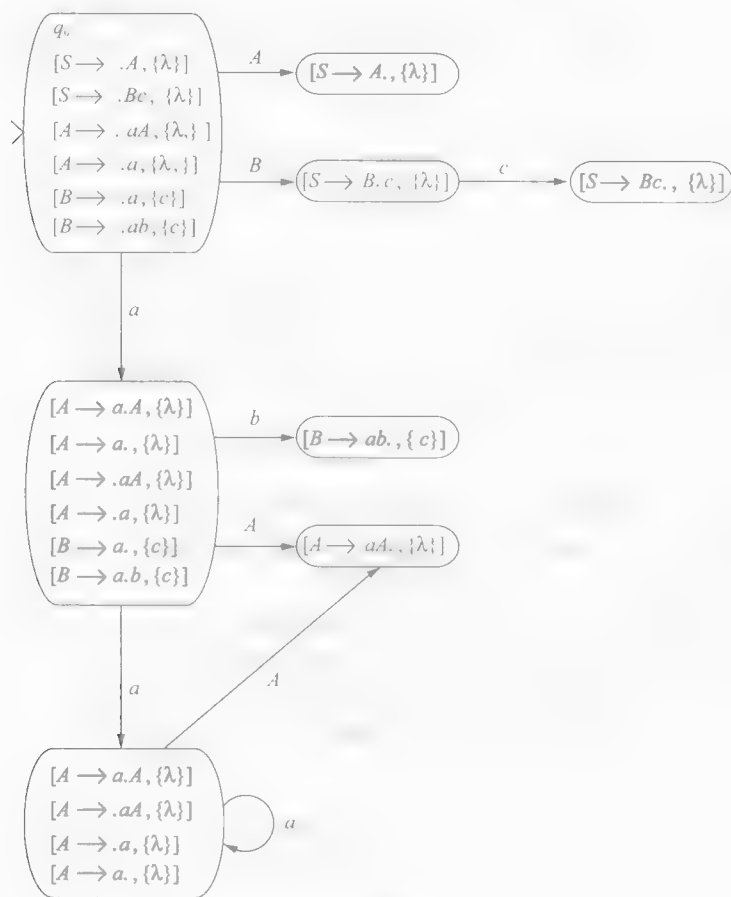


图 20-5 G 的确定型 LR(1) 机

文法 G 的形式如下所示:

$$\begin{aligned} G: S &\rightarrow A \mid Bc \\ A &\rightarrow aA \mid a \\ B &\rightarrow a \mid ab \end{aligned}$$

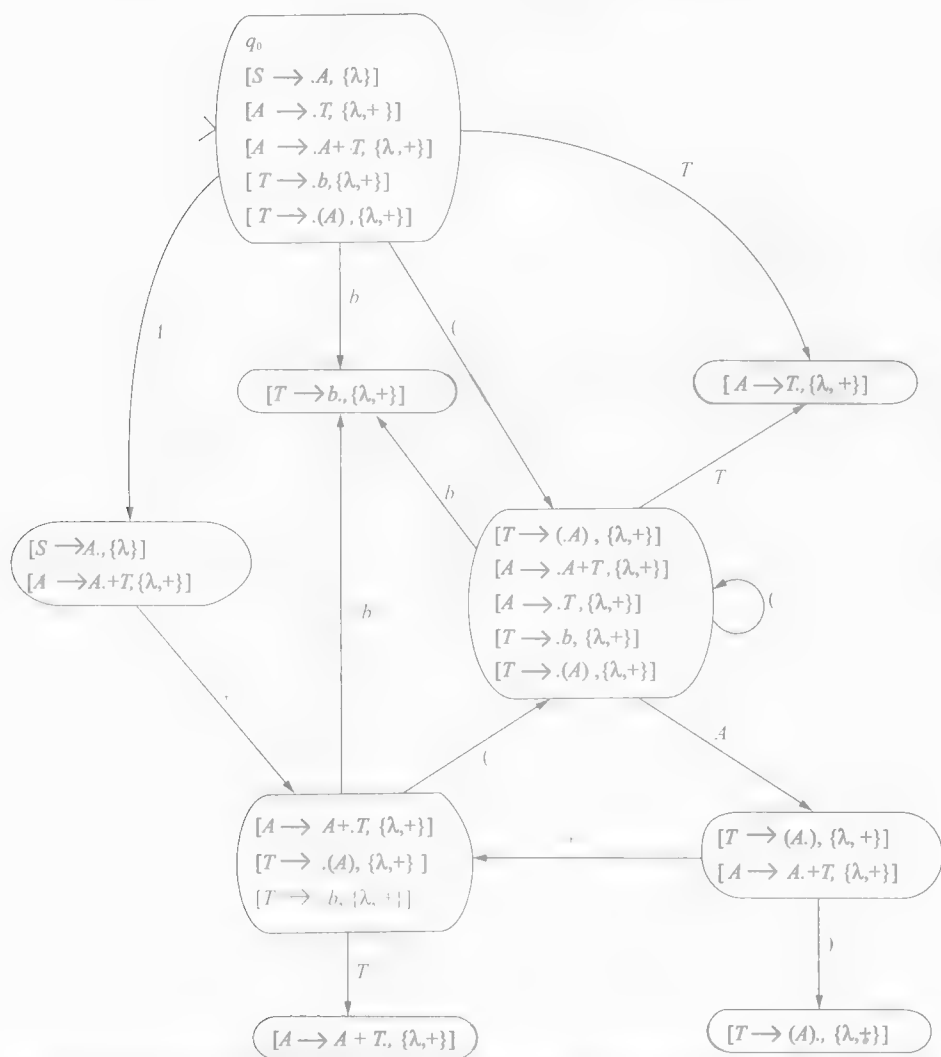
如果分析器的动作是由使用单一预读符号来惟一确定的, 那么文法是 LR(1) 的。确定型 LR(1) 机的结构可以用于定义 LR(1) 文法。

[616] 定义 20.5.3 设 G 是具有非递归开始符号的上下文无关文法。如果 G 的确定型 LR(1) 机的扩展状态转换函数满足下述情况, 那么文法 G 是 LR(1) 的:

i) 如果 $\hat{\delta}(q, u)$ 包含了完全项 $[A \rightarrow w., \{z_1, z_2, \dots, z_n\}]$, 而且 $\hat{\delta}(q, u)$ 包含了项 $[B \rightarrow r.as, \{y_1, y_2, \dots, y_k\}]$, 那么, 对于所有的 $1 \leq i \leq n, a \neq z_i$ 。

ii) 如果 $\hat{\delta}(q, u)$ 包含了两个完全项 $[A \rightarrow w., \{z_1, z_2, \dots, z_n\}]$ 和 $[B \rightarrow v., \{y_1, y_2, \dots, y_k\}]$, 那么, 对于所有的 $1 \leq i \leq k$ 和 $1 \leq j \leq n, y_i \neq z_j$ 。

例 20.5.1 确定型的 LR(1) 机是为文法 AE 构造的。包含完全项 $S \rightarrow A.$ 的状态也包含了 $A \rightarrow A. + T$ 。这样看来, AE 就不是 LR(0) 的。一旦进入这个状态, LR(1) 分析器就会因失败而停机, 知道预读符号是 + 或者空串。在后面这种情况中, 整个输入串都已经被读入, 并且指明了使用规则 $S \rightarrow A$ 的归约。当预读符号是 + 的时候, 分析器发生转换, 以尝试构造串 $A + T$ 。□



LR(1) 文法分析器的扫描串 u 的动作是由计算 $\delta(q, u)$ 的结果来决定的。算法 20.5.4 给出了用于 [617] 分析 LR(1) 文法的确定型算法。

算法 20.5.4

用于 LR(1) 文法的分析器

输入: LR(1) 文法 $G = (V, \Sigma, P, S)$

字符串 $p \in \Sigma^*$

G 的确定型 LR(1) 机器

1. Let $p = zv$, 其中 $z \in \Sigma \cup \{\lambda\}$ 且 $v \in \Sigma^*$
(z 是向前的符号, v 是输入的提示符)

2. $u := \lambda$

3. dead-end: $= false$

4. repeat

4.1. if $\hat{\delta}(q, u)$ 包含 $[A \rightarrow w \cdot, \{z_1, \dots, z_n\}]$

其中 $u = xw$ 且对于某个 $1 \leq i \leq n$ 有 $z = z_i$ then $u := xA$

```

else if  $z \neq \lambda$  且  $\hat{\delta}(q_s, u)$  包含一个项  $A \rightarrow p.zq$  then
    (转移并获得新的向前的符号)
    4. 1. 1.  $u := uz$ 
    4. 1. 2. 设  $v = zv'$ , 其中  $z \in \Sigma \cup \{\lambda\}$  且  $v' \in \Sigma^*$ 
    4. 1. 3.  $v := v'$ 
end if
else dead - end: = true
until  $u = S$  or dead - end
5. if  $u = S$  then 接收 else 拒绝

```

[618]

对于 LR(1) 文法来说, LR(1) 机的结构保证了步骤 4.1 指定的动作是惟一的, 当一个状态包含了多于一个完全项的时候, 预读符号就会指明合适的操作。

例 20.5.2 算法 20.5.4 和图 20-5 中的确定型 LR(1) 机使用下述文法分析串 aaa 和 ac :

$G: S \rightarrow A \mid Bc$

$A \rightarrow aA \mid a$

$B \rightarrow a \mid ab.$

u	z	v	计算	行为
λ	a	aa	$\hat{\delta}(q_s, \lambda) =$	转移
			$\{ [S \rightarrow A, \{\lambda\}],$ $[S \rightarrow Bc, \{\lambda\}],$ $[A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}],$ $[B \rightarrow a \mid c],$ $[B \rightarrow ab \mid c]]$	
a	a	a	$\hat{\delta}(q_s, a) =$	转移
			$\{ [A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}],$ $[A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}],$ $[B \rightarrow a, \{\lambda\}],$ $[B \rightarrow ab, \{\lambda\}]]$	
aa	a	λ	$\hat{\delta}(q_s, aa) =$	转移
			$\{ [A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}]]$	
aaa	λ	λ	$\hat{\delta}(q_s, aaa) =$	归约
			$\{ [A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow aA, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}],$ $[A \rightarrow a, \{\lambda\}]]$	
aaA	λ	λ	$\hat{\delta}(q_s, aaA) =$	归约
			$\{ [A \rightarrow aA, \{\lambda\}]]$	
aA	λ	λ	$\hat{\delta}(q_s, aA) =$	归约
			$\{ [A \rightarrow aA, \{\lambda\}]]$	
A	λ	λ	$\hat{\delta}(q_s, A) =$	归约
			$\{ [S \rightarrow A, \{\lambda\}]]$	

(续)

u	z	v	计算	行为
S				
λ	a	c	$\hat{\delta}(q_s, \lambda) =$	转移
				$\{[S \rightarrow \cdot A, \{\lambda\}],$ $[S \rightarrow \cdot Bc, \{\lambda\}],$ $[A \rightarrow \cdot aA, \{\lambda\}],$ $[A \rightarrow \cdot a, \{\lambda\}],$ $[B \rightarrow \cdot a c],$ $[B \rightarrow \cdot ab c]\}$
a	c	λ	$\hat{\delta}(q_s, a) =$	归约
				$\{[A \rightarrow a \cdot A, \{\lambda\}],$ $[A \rightarrow a \cdot, \{\lambda\}],$ $[A \rightarrow \cdot aA, \{\lambda\}],$ $[A \rightarrow \cdot a, \{\lambda\}],$ $[B \rightarrow a \cdot, \{c\}],$ $[B \rightarrow a \cdot b, \{c\}]\}$
B	c	λ	$\hat{\delta}(q_s, B) =$	转移
Bc	λ	λ	$\hat{\delta}(q_s, Bc) =$	归约
S				

619

20.6 练习

1. 给出下属文法规则的 LR(0) 上下文、建立非确定型 LR(0) 机、利用这个构造确定型 LR(0) 机 文法是 LR(0) 的吗?

- a) $S \rightarrow AB$
 $A \rightarrow aA \mid b$
 $B \rightarrow bB \mid a$
- b) $S \rightarrow Ac$
 $A \rightarrow BA \mid \lambda$
 $B \rightarrow aB \mid b$
- c) $S \rightarrow A$
 $A \rightarrow aAb \mid bAa \mid \lambda$
- d) $S \rightarrow aA \mid AB$
 $A \rightarrow aAb \mid b$
 $B \rightarrow ab \mid b$
- e) $S \rightarrow BA \mid bAB$
 $A \rightarrow aA \mid \lambda$
 $B \rightarrow Bb \mid b$
- f) $S \rightarrow A \mid aB$
 $A \rightarrow BC \mid \lambda$
 $B \rightarrow Bb \mid C$
 $C \rightarrow Cc \mid c$

2. 建立下述文法的确定型 LR(0) 机。

$S \rightarrow aAb \mid aB$
 $A \rightarrow Aa \mid \lambda$
 $B \rightarrow Ac.$

620

利用例 20.3.1 中描述的技术来跟踪串 $aaab$ 和串 ac 的分析。

3. 证明没有末端标记的文法 AE 不是 LR(0) 的。
4. 证明引理 20.4.2。
5. 证明 LR(0) 文法是无二义性的。
6. 定义规则 $A \rightarrow w$ 的 LR(k) 上下文。
7. 对于下述每一个文法，构造其非确定型和确定型 LR(1) 机。该文法是 LR(1) 的吗?

- a) $S \rightarrow Ac$
 $A \rightarrow BA \mid \lambda$
 $B \rightarrow aB \mid b$
- b) $S \rightarrow A$
 $A \rightarrow AaAb \mid \lambda$

- c) $S \rightarrow A$
 $A \rightarrow aAb \mid B$
 $B \rightarrow Bb \mid b$
- d) $S \rightarrow A$
 $A \rightarrow BB$
 $B \rightarrow aB \mid b$

- e) $S \rightarrow A$
 $A \rightarrow AAa \mid AAb \mid c$

8. 为例 20.4.3 中引入的文法构造 LR(1) 机。该文法是 LR(1) 的吗?

9. 利用 LR(1) 分析器和文法 AE 分析下述串。使用例 20.5.2 中给出的格式跟踪分析器的动作。AE 的确定型 LR(1) 机在例 20.5.1 中给出。

- a) $b + b$
b) (b)
c) $b + + b$

参考文献注释

LR 文法在 Knuth [1965] 中引入。LR 机中的状态移进和状态转换数对于计算机语言分析来说是不实际的。Korenjak [1969] 和 De Remer [1969, 1971] 开发出了简化方法, 从而消除了这些困难。后边的工作引入了 SLR (简单 LR) 和 LALR (预读 LR) 文法。LR(k) 文法类和可以被确定型分析的其他类文法 (包括 LR(k) 文法) 之间的关系在 Aho 和 Ullman [1972, 1973] 中得到阐述。

621
622

附录 I 标记索引

符号	页码 [⊙]	解释
\in	8	是……的元素
\notin	8	不是……的元素
$\{x \mid \dots\}$	8	x 的集合
\mathbf{N}	8	自然数集合
\emptyset	8	空集合
\subseteq	8	是……的子集
$\mathcal{P}(X)$	9	是……的幂集
\cup	9	并
\cap	9	交
$-$	9	$X - Y$: 差
\overline{X}	9	补
\times	11	$X \times Y$: 笛卡儿积
$[x, y]$	11	有序对
$f: X \rightarrow Y$	12	f 是从 X 到 Y 的函数
$f(x)$	12	x 的由函数 f 赋予的值
$f(x) \uparrow$	13	$f(x)$ 未定义
$f(x) \downarrow$	13	$f(x)$ 已定义
div	13	整数除法
\equiv	15	等价关系
$[]_m$	15	等价类
$card$	16	集的势
s	24, 300	后继函数
$\sum_{i=1}^m$	31, 398	有界总和
$\dot{-}$	39, 395	真减
λ	42	空串
Σ^*	42	Σ 上的串集合
$length$	43	串的长度
uv	44	u 和 v 的连接
u^n	44	n 个 u 连接
u^R	45	u 的逆
XY	47	集合 X 和集合 Y 连接
X^i	47	将 i 个 X 连接起来
X^*	48	X 上的串
X^+	48	X 上的非空串
∞	48	无穷
\emptyset	50	空集的正则表达式
λ	50	空串的正则表达式
a	50	集合 $\{a\}$ 的正则表达式

⊙ 本页码所指的是本书页边栏中标注的页码。——编辑注

(续)

符号	页码	解释
\cup	50	正则表达式的并操作
\rightarrow	65, 69, 326	文法的规则
\Rightarrow	67, 69, 326	应用一个规则进行推导
$\stackrel{*}{\Rightarrow}$	69, 326	由...推导的
$\stackrel{+}{\Rightarrow}$	69	使用一个或者多个规则进行推导的
$\stackrel{n}{\Rightarrow}$	69	使用 n 个规则进行推导
$L(G)$	70, 326	文法 G 的语言
$n_x(u)$	84	x 在 u 中出现的次数
\Rightarrow_L	91	最左规则应用
\Rightarrow_R	91	最右规则应用
A_{opt}	94, 631	A 的出现是可选择的
δ	147, 163, 222, 256	状态转换函数
$L(M)$	148, 163, 234, 260	M 机的语言
\vdash	149, 224, 258	由一个状态转换生成
\vdash^*	149, 224, 258	由零个或者多个状态转换生成
$\hat{\delta}$	151, 185	扩展的状态转换函数
λ -closure	170	λ -闭包函数
Γ	222, 256	栈或者带字母表
B	256	空带符号
lo	286	字典顺序
χ_L	298	语言 L 的特征函数
$\hat{\chi}_L$	298	语言 L 的部分特征函数
\bar{i}	299, 471	i 的表示法
z	300, 390	零函数
e	300	空函数
$p_i^{(k)}$	300, 390	k 个变量的投影函数
id	301	恒等函数
$pred$	301	前驱函数
\circ	308	构成
$c_i^{(k)}$	311, 391	具有 k 个符号的常函数
$\lfloor x \rfloor$	320	小于等于 x 的最大整数
P	343	判定问题
U	356	通用图灵机
L_H	357, 365	停机问题的语言
F	372	递归可枚举语言的特性
$!$	393	阶
$\prod_{i=0}^n$	398	有界乘积
$\mu z[p]$	400, 413	无界限最小值
$\overset{y}{\mu} z[p]$	401	有界最小值
quo	404	商函数
$pn(i)$	405	计算第 i 个素数的函数
gn_k	406	$k+1$ 个变量的哥德尔数函数
$dec(i, x)$	407	解码函数
gn_f	408	有界的哥德尔数函数
tr_M	417, 420	图灵机跟踪函数
\mathcal{P}	431, 468	多项式语言的类

(续)

符号	页码	解释
\mathcal{NP}	431, 469	非确定的多项式语言的类
$O(g)$	436	函数 g 的大 O 表示法
$\Theta(g)$	438	函数 g 的大 θ 表示法
$ i $	438	i 的绝对值
t_{CM}	443	时间复杂性函数
$\lceil x \rceil$	451	大于等于 x 的最小整数
$rep(p)$	471	问题实例 p 的表达式
\wedge	481	合取
\vee	481	析取
\neg	481	非
L_{SAT}	483	可满足性问题的语言
\mathcal{NPC}	492	\mathcal{NP} 完全问题的类
$\text{Co-}\mathcal{NP}$	531	\mathcal{NP} 的补
s_{CM}	532	空间复杂性函数
\inf	538	下确界, 最大的受限下界值
$\mathcal{P}\text{-Space}$	540	多项式空间语言的类
$\mathcal{NP}\text{-Space}$	540	非确定的多项式空间语言的类
$g(G)$	556	文法 G 的图
$LA(A)$	572	变量 A 的预读集合
$LA(A \rightarrow w)$	572	规则 $A \rightarrow w$ 的预读集合
$trunc_k$	575	长度为 k 的切断函数
$FIRST_k(u)$	576	串 u 的 $FIRST_k$ 集合
$FOLLOW_k(A)$	577	变量 A 的 $FOLLOW_k$ 集合
$shift$	599	移进函数

附录 II 希腊字母表

大写	小写	名称
Α	α	alpha
Β	β	beta
Γ	γ	gamma
Δ	δ	delta
Ε	ϵ	epsilon
Ζ	ζ	zeta
Η	η	eta
Θ	θ	theta
Ι	ι	iota
Κ	κ	kappa
Λ	λ	lambda
Μ	μ	mu
Ν	ν	nu
Ξ	ξ	xi
Ο	\omicron	omicron
Π	π	pi
Ρ	ρ	rho
Σ	σ	sigma
Τ	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
Χ	χ	chi
Ψ	ψ	psi
Ω	ω	omega

附录Ⅲ ASCⅡ字符集

美国标准代码是用于信息转换的，该标准码的一种更为普遍的引用方式是 ASCⅡ码。这种代码可以表示可印刷符号以及使用数字 0 到 127 的二进制表示法的特殊功能。数字 0 到 31 是控制字符，并且该栏标记的名字给出了与字符相关的动作的缩写。例如，数字 14 和 15 表示如果遇到这样的字符，印刷字符应该从新一行开始（LF，换行）或者从新一页开始（FF，换页）。数字 32（空白区）到 126 已经作为文本文档的标准编码而得到广泛接收。

代码	等号	名称	代码	等号	代码	等号	代码	等号
0	^@	NUL	32		64	@	96	`
1	^A	SOH	33	!	65	A	97	a
2	^B	STX	34	"	66	B	98	b
3	^C	ETX	35	#	67	C	99	c
4	^D	EOT	36	\$	68	D	98	d
5	^E	ENQ	37	%	69	E	101	e
6	^F	ACK	38	&	70	F	102	f
7	^G	BEL	39	'	71	G	103	g
8	^H	BS	40	(72	H	104	h
9	^I	TAB	41)	73	I	105	i
10	^J	LF	42	*	74	J	106	j
11	^K	VT	43	+	75	K	107	k
12	^L	FF	44	,	76	L	108	l
13	^M	CR	45	-	77	M	109	m
14	^N	SO	46	.	78	N	110	n
15	^O	SI	47	/	79	O	111	o
16	^P	DLE	48	0	80	P	112	p
17	^Q	DC1	49	1	81	Q	113	q
18	^R	DC2	50	2	82	R	114	r
19	^S	DC3	51	3	83	S	115	s
20	^T	DC4	52	4	84	T	116	t
21	^U	NAK	53	5	85	U	117	u
22	^V	SYN	54	6	86	V	118	v
23	^W	ETB	55	7	87	W	119	w
24	^X	CAN	56	8	88	X	120	x
25	^Y	EM	57	9	89	Y	121	y
26	^Z	SUB	58	:	90	Z	122	z
27	^[ESC	59	;	91	[123	{
28	^\	FS	60	<	92	\	124	
29	^_	GS	61	=	93]	125	}
30	^^	RS	62	>	94	^	126	~
31	^-	US	63	?	95	_	127	DEL

附录IV Java 的 BNF 范式定义

Java 程序设计语言是在 Sun Microsystems 公司的 James Gosling 的指导下开发出来的。Java 于 1995 年问世, 该语言平台独立, 是一种面向对象的编程语言, 对于 Internet 和网络应用程序尤为适用。自此, Java 变成了 Internet 应用程序最为通用的语言之一。

Java 语言的语法是从 Gosling 等人定义的 BNF 范式推导出来的[2000]。除了在符号上标有 *opt* 作为下标, 以保留终结符或者变量的名称作为选项之外, 规则已经被转换成标准的上下文无关符号。*opt* 的使用减少了所需规则的数目, 但是带有可选构件的规则可以很容易地被转换成等价的上下文无关文法。右边带有变量 B_{opt} 的规则可以被两个规则代替。其中一个是将 B_{opt} 用 B 来代替, 然后它在另一个规则中被删除。例如, $A \rightarrow B_{opt}C$ 用 $A \rightarrow BC$ $C \rightarrow B$ 来代替。 n 个带有下标 *opt* 的符号可以创建 2^n 个上下文无关的规则。文法的开始符号是变量 $\langle \text{编译单元} \rangle$ 。

1. $\text{CompilationUnit} \rightarrow \langle \text{PackageDeclaration} \rangle_{opt} \langle \text{ImportDeclarations} \rangle_{opt} \langle \text{TypeDeclarations} \rangle_{opt}$

Declarations

2. $\langle \text{ImportDeclarations} \rangle \rightarrow \langle \text{ImportDeclarations} \rangle | \langle \text{ImportDeclarations} \rangle \langle \text{ImportDeclaration} \rangle$
3. $\langle \text{TypeDeclarations} \rangle \rightarrow \langle \text{TypeDeclaration} \rangle | \langle \text{TypeDeclarations} \rangle \langle \text{TypeDeclaration} \rangle$
4. $\langle \text{PackageDeclaration} \rangle \rightarrow \text{package } \langle \text{PackageName} \rangle ;$
5. $\langle \text{ImportDeclaration} \rangle \rightarrow \langle \text{SingleTypeImportDeclaration} \rangle | \langle \text{TypeImportOnDemand} \rangle$
6. $\langle \text{SingleTypeImportDeclaration} \rangle \rightarrow \text{import } \langle \text{TypeName} \rangle ;$
7. $\langle \text{TypeImportOnDemandDeclaration} \rangle \rightarrow \text{import } \langle \text{PackageName} \rangle . * ;$
8. $\langle \text{TypeDeclaration} \rangle \rightarrow \langle \text{ClassDeclaration} \rangle | \langle \text{Declaration} \rangle | ;$
9. $\langle \text{Type} \rangle \rightarrow \langle \text{PrimitiveType} \rangle \langle \text{ReferenceType} \rangle$
10. $\langle \text{PrimitiveType} \rangle \rightarrow \langle \text{NumericType} \rangle \text{ boolean}$
11. $\langle \text{NumericType} \rangle \rightarrow \langle \text{IntegralType} \rangle | \langle \text{FloatingPointType} \rangle$
12. $\langle \text{IntegralType} \rangle \rightarrow \text{byte} | \text{short} | \text{int} | \text{long} | \text{char}$
13. $\langle \text{FloatingPointType} \rangle \rightarrow \text{float} | \text{double}$

Reference Types and Values

14. $\langle \text{ReferenceType} \rangle \rightarrow \langle \text{ClassOrInterfaceType} \rangle | \langle \text{ArrayType} \rangle$
15. $\langle \text{ClassOrInterfaceType} \rangle \rightarrow \langle \text{ClassType} \rangle | \langle \text{InterfaceType} \rangle$
16. $\langle \text{ClassType} \rangle \rightarrow \langle \text{TypeName} \rangle$
17. $\langle \text{InterfaceType} \rangle \rightarrow \langle \text{TypeName} \rangle$
18. $\langle \text{ArrayType} \rangle \rightarrow \langle \text{Type} \rangle []$

Class Declarations

19. $\langle \text{ClassDeclaration} \rangle \rightarrow \langle \text{ClassModifier} \rangle_{opt} \text{class } \langle \text{Identifier} \rangle \langle \text{Super} \rangle_{opt} \langle \text{Interfaces} \rangle_{opt} \langle \text{Classbody} \rangle$
20. $\langle \text{ClassModifiers} \rangle \rightarrow \langle \text{ClassModifier} \rangle | \langle \text{ClassModifiers} \rangle \langle \text{ClassModifier} \rangle$
21. $\langle \text{ClassModifier} \rangle \rightarrow \text{public} | \text{abstract} | \text{final}$
22. $\langle \text{Super} \rangle \rightarrow \text{extends } \langle \text{ClassType} \rangle$
23. $\langle \text{Interfaces} \rangle \rightarrow \text{implements } \langle \text{InterfaceTypeList} \rangle$

24. $\langle \text{InterfaceTypeList} \rangle \rightarrow \langle \text{InterfaceType} \rangle \mid \langle \text{InterfaceTypeList} \rangle \langle \text{InterfaceType} \rangle$
25. $\langle \text{ClassBody} \rangle \rightarrow \mid \langle \text{ClassBodyDeclarations} \rangle_{\text{opt}} \mid$
26. $\langle \text{ClassBodyDeclarations} \rangle \rightarrow \langle \text{ClassBodyDeclaration} \rangle \mid$
 $\langle \text{ClassBodyDeclaration} \rangle \langle \text{ClassBodyDeclarations} \rangle$
27. $\langle \text{ClassBodyDeclaration} \rangle \rightarrow \langle \text{ClassMemberDeclaration} \rangle \mid \langle \text{StaticInitializer} \rangle \mid$
 $\langle \text{ConstructorDeclarations} \rangle$
28. $\langle \text{ClassMemberDeclaration} \rangle \rightarrow \langle \text{FieldDeclaration} \rangle \mid \langle \text{MethodDeclaration} \rangle$
- Field Declarations**
29. $\langle \text{FieldDeclaration} \rangle \rightarrow \langle \text{FieldModifiers} \rangle_{\text{opt}} \langle \text{Type} \rangle \langle \text{VariableDeclarators} \rangle ;$
30. $\langle \text{VariableDeclarators} \rangle \rightarrow \langle \text{VariableDeclarator} \rangle \mid \langle \text{VariableDeclarators} \rangle ,$
 $\langle \text{VariableDeclarator} \rangle$
31. $\langle \text{VariableDeclarator} \rangle \rightarrow \langle \text{VariableDeclaratorID} \rangle \mid$
 $\langle \text{VariableDeclaratorsID} \rangle = \langle \text{VariableInitializer} \rangle$
32. $\langle \text{VariableDeclaratorID} \rangle \rightarrow \langle \text{Identifier} \rangle \mid \langle \text{VariableDeclaratorsID} \rangle []$
33. $\langle \text{VariableInitializer} \rangle \rightarrow \langle \text{Expression} \rangle \mid \langle \text{ArrayInitializer} \rangle$
34. $\langle \text{FieldModifiers} \rangle \rightarrow \langle \text{FieldModifier} \rangle \mid \langle \text{FieldModifiers} \rangle \langle \text{FieldModifier} \rangle$
35. $\langle \text{FieldModifier} \rangle \rightarrow \text{public} \mid \text{protected} \mid \text{private} \mid \text{final} \mid \text{static} \mid \text{transient} \mid \text{volatile}$
- Method Declarations**
36. $\langle \text{MethodDeclaration} \rangle \rightarrow \langle \text{MethodHeader} \rangle \langle \text{MethodBody} \rangle$
37. $\langle \text{MethodHeader} \rangle \rightarrow \langle \text{MethodModifiers} \rangle_{\text{opt}} \langle \text{ResultType} \rangle \langle \text{MethodDeclarator} \rangle$
 $\langle \text{Throws} \rangle_{\text{opt}}$
38. $\langle \text{ResultType} \rangle \rightarrow \langle \text{Type} \rangle \mid \text{void}$
39. $\langle \text{MethodDeclarator} \rangle \rightarrow \langle \text{Identifier} \rangle (\langle \text{FormalParameterList} \rangle_{\text{opt}})$
 $\langle \text{MethodDeclarator} \rangle []$
40. $\langle \text{FormalParameterList} \rangle \rightarrow \langle \text{FormalParameter} \rangle \mid$
 $\langle \text{FormalParameterList} \rangle \langle \text{FormalParameter} \rangle$
41. $\langle \text{FormalParameter} \rangle \rightarrow \langle \text{Type} \rangle \langle \text{VariableDeclaratorID} \rangle$
42. $\langle \text{MethodModifiers} \rangle \rightarrow \langle \text{MethodModifier} \rangle \mid \langle \text{MethodModifiers} \rangle \langle \text{MethodModifiers} \rangle$
43. $\langle \text{MethodModifier} \rangle \rightarrow \text{public} \mid \text{protected} \mid \text{private} \mid \text{abstract} \mid \text{final} \mid$
 $\text{static} \mid \text{synchronized} \mid \text{native}$
44. $\langle \text{Throws} \rangle \rightarrow \text{throws} \langle \text{ClassTypeList} \rangle$
45. $\langle \text{ClassTypeList} \rangle \rightarrow \langle \text{ClassType} \rangle \mid \langle \text{ClassTypeList} \rangle , \langle \text{ClassType} \rangle$
46. $\langle \text{MethodBody} \rangle \rightarrow \langle \text{Block} \rangle \mid ;$
- Constructor Declarations**
47. $\langle \text{ConstructorDeclaration} \rangle \rightarrow \langle \text{ConstructorModifiers} \rangle_{\text{opt}} \langle \text{ConstructorDeclarator} \rangle$
 $\langle \text{Throws} \rangle_{\text{opt}} \langle \text{ConstructorBody} \rangle$
48. $\langle \text{ConstructorDeclarator} \rangle \rightarrow \langle \text{SimpleTypeName} \rangle (\langle \text{FormalParameterList} \rangle_{\text{opt}})$
49. $\langle \text{ConstructorModifiers} \rangle \rightarrow \langle \text{ConstructorModifier} \rangle \mid$
 $\langle \text{ConstructorModifiers} \rangle \langle \text{ConstructorModifier} \rangle$
50. $\langle \text{ConstructorModifier} \rangle \rightarrow \text{public} \mid \text{private} \mid \text{protected}$
51. $\langle \text{ConstructorBody} \rangle \rightarrow \{ \langle \text{ExplicitConstructorInvocation} \rangle_{\text{opt}} \langle \text{BlockStatements} \rangle_{\text{opt}} \}$
52. $\langle \text{ExplicitConstructorInvocation} \rangle \rightarrow \text{this} (\langle \text{ArgumentList} \rangle_{\text{opt}}) ; \text{mid}$
 $\text{super} (\langle \text{ArgumentList} \rangle_{\text{opt}}) ;$
- Interface Declarations**

633

53. $\langle \text{InterfaceDeclaration} \rangle \rightarrow \langle \text{InterfaceModifiers} \rangle_{\text{opt}} \text{interface } \langle \text{Identifier} \rangle$
 $\langle \text{ExtendsInterface} \rangle_{\text{opt}} \langle \text{InterfaceBody} \rangle$
54. $\langle \text{InterfaceModifiers} \rangle \rightarrow \langle \text{InterfaceModifier} \rangle \langle \text{InterfaceModifiers} \rangle \langle \text{InterfaceModifier} \rangle$
55. $\langle \text{InterfaceModifier} \rangle \rightarrow \text{public} \mid \text{abstract}$
56. $\langle \text{ExtendsInterfaces} \rangle \rightarrow \text{extends } \langle \text{InterfaceType} \rangle \mid$
 $\langle \text{ExtendsInterfaces} \rangle, \langle \text{InterfaceType} \rangle$
57. $\langle \text{InterfaceBody} \rangle \rightarrow \{ \langle \text{InterfaceMemberDeclaration} \rangle_{\text{opt}} \}$
58. $\langle \text{InterfaceMemberDeclarations} \rangle \rightarrow \langle \text{InterfaceMemberDeclaration} \rangle \mid$
 $\langle \text{InterfaceMemberDeclarations} \rangle \langle \text{InterfaceMemberDeclaration} \rangle$
59. $\langle \text{InterfaceMemberDeclaration} \rangle \rightarrow \langle \text{ConstantDeclaration} \rangle \mid$
 $\langle \text{AbstractMethodDeclaration} \rangle$

Constant Declarations

60. $\langle \text{ConstantDeclaration} \rangle \rightarrow \langle \text{ConstantModifiers} \rangle_{\text{opt}} \langle \text{Type} \rangle \langle \text{VariableDeclarator} \rangle$
61. $\langle \text{ConstantModifiers} \rangle \rightarrow \text{public} \mid \text{static} \mid \text{final}$

Abstract Method Declarations

62. $\langle \text{AbstractMethodDeclaration} \rangle \rightarrow \langle \text{AbstractMethodModifiers} \rangle_{\text{opt}} \langle \text{ResultType} \rangle$
 $\langle \text{MethodDeclarator} \rangle \langle \text{Throws} \rangle_{\text{opt}}$
63. $\langle \text{AbstractMethodModifiers} \rangle \rightarrow \langle \text{AbstractMethodModifier} \rangle \mid$
 $\langle \text{AbstractMethodModifiers} \rangle \langle \text{AbstractMethodModifier} \rangle$
64. $\langle \text{AbstractMethodModifier} \rangle \rightarrow \text{public} \mid \text{abstract}$

Array Initializers

65. $\langle \text{ArrayInitializer} \rangle \rightarrow \{ \langle \text{Variable Initializers} \rangle_{\text{opt}, \text{opt}} \}$
66. $\langle \text{VariableInitializers} \rangle \rightarrow \langle \text{VariableInitializer} \rangle \mid$
 $\langle \text{VariableInitializers} \rangle \langle \text{VariableInitializers} \rangle$

Blocks and Local Variable Declaration

67. $\langle \text{Block} \rangle \rightarrow \{ \langle \text{BlockStatements} \rangle_{\text{opt}} \}$
68. $\langle \text{BlockStatements} \rangle \rightarrow \langle \text{BlockStatement} \rangle \mid \langle \text{BlockStatements} \rangle \langle \text{BlockStatement} \rangle$
69. $\langle \text{BlockStatement} \rangle \rightarrow \langle \text{LocalVariableDeclarationStatement} \rangle \mid \langle \text{Statement} \rangle$
70. $\langle \text{StaticInitializer} \rangle \rightarrow \text{static } \langle \text{Block} \rangle$
71. $\langle \text{LocalVariableDeclarationStatement} \rangle \rightarrow \langle \text{LocalVariableDeclaration} \rangle$
72. $\langle \text{LocalVariableDeclaration} \rangle \rightarrow \langle \text{Type} \rangle \langle \text{VariableDeclarators} \rangle$

Statements

73. $\langle \text{Statement} \rangle \rightarrow \langle \text{StatementWithoutTrailingSubstatement} \rangle \mid \langle \text{LabeledStatement} \rangle \mid$
 $\langle \text{IfThenStatement} \rangle \mid \langle \text{IfThenElseStatement} \rangle \mid$
 $\langle \text{WhileStatement} \rangle \mid \langle \text{ForStatement} \rangle$
74. $\langle \text{StatementNoShortIf} \rangle \rightarrow \langle \text{StatementWithoutTrailingSubstatement} \rangle \mid$
 $\langle \text{LabeledStatementNoShortIf} \rangle \mid$
 $\langle \text{IfThenStatementNoShortIf} \rangle \mid$
 $\langle \text{IfThenElseStatementNoShortIf} \rangle \mid$
 $\langle \text{ForStatementNoShortIf} \rangle$
75. $\langle \text{StatementWithoutTrailingSubstatement} \rangle \rightarrow \langle \text{Block} \rangle$
 $\langle \text{EmptyStatement} \rangle \mid \langle \text{ExpressionStatement} \rangle \mid$
 $\langle \text{SwitchStatement} \rangle \mid \langle \text{DoStatement} \rangle \mid$
 $\langle \text{BreakStatement} \rangle \mid \langle \text{Continue Statement} \rangle \mid$

634

$\langle \text{ReturnStatement} \rangle \mid \langle \text{SynchronizedStatement} \rangle \mid$
 $\langle \text{ThrowStatement} \rangle \mid \langle \text{TryStatement} \rangle$

Empty, Labeled, and Expression Statements

76. $\langle \text{EmptyStatement} \rangle \rightarrow ;$
77. $\langle \text{LabeledStatement} \rangle \rightarrow \langle \text{Identifier} \rangle : \langle \text{Statement} \rangle$
78. $\langle \text{LabeledStatementNoShortIf} \rangle \rightarrow \langle \text{Identifier} \rangle : \langle \text{StatementNoShortIf} \rangle$
79. $\langle \text{ExpressionStatement} \rangle \rightarrow \langle \text{StatementExpression} \rangle ;$
80. $\langle \text{StatementExpression} \rangle \rightarrow \langle \text{Assignment} \rangle \mid \langle \text{PreincrementExpression} \rangle \mid$
 $\langle \text{PredecrementExpression} \rangle \mid \langle \text{PostincrementExpression} \rangle \mid$
 $\langle \text{PostdecrementExpression} \rangle \mid \langle \text{MethodInvocation} \rangle \mid$
 $\langle \text{ClassInstanceCreationExpression} \rangle$

If Statements

81. $\langle \text{IfThenStatement} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{Statement} \rangle$
82. $\langle \text{IfThenElseStatement} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle \text{ else } \langle \text{Statement} \rangle$
83. $\langle \text{IfThenElseStatementNoShortIf} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle$
 $\text{ else } \langle \text{StatementNoShortIf} \rangle$

Switch Statement

84. $\langle \text{SwitchStatement} \rangle \rightarrow \text{switch} (\langle \text{Expression} \rangle) \langle \text{SwitchBlock} \rangle$
85. $\langle \text{SwitchBlock} \rangle \rightarrow \{ \langle \text{SwitchBlockStatementGroups} \rangle_{\text{opt}} \langle \text{SwitchLabel} \rangle_{\text{opt}} \}$
86. $\langle \text{SwitchBlockStatementGroups} \rangle \rightarrow \langle \text{SwitchBlockStatementGroup} \rangle \mid$
 $\langle \text{SwitchBlockStatementGroups} \rangle \langle \text{SwitchBlockStatementGroups} \rangle$
87. $\langle \text{SwitchBlockStatementGroup} \rangle \rightarrow \langle \text{SwitchLabels} \rangle \langle \text{BlockStatements} \rangle$
88. $\langle \text{SwitchLabels} \rangle \rightarrow \langle \text{SwitchLabel} \rangle \mid \langle \text{SwitchLabels} \rangle \langle \text{SwitchLabel} \rangle$
89. $\langle \text{SwitchLabel} \rangle \rightarrow \text{case } \langle \text{ConstantExpression} \rangle : \mid \text{default} :$

While, Do, and For Statements

90. $\langle \text{WhileStatement} \rangle \rightarrow \text{while} (\langle \text{Expression} \rangle) \langle \text{Statement} \rangle$
91. $\langle \text{WhileStatementNoShortIf} \rangle \rightarrow \text{while} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle$
92. $\langle \text{DoStatement} \rangle \rightarrow \text{do } \langle \text{Statement} \rangle \text{ while } (\langle \text{Expression} \rangle) ;$
93. $\langle \text{ForStatement} \rangle \rightarrow \text{for} (\langle \text{ForInit} \rangle_{\text{opt}} ; \langle \text{Expression} \rangle_{\text{opt}} ; \langle \text{ForUpdate} \rangle_{\text{opt}}) \langle \text{Statement} \rangle$
94. $\langle \text{ForStatementNoShortIf} \rangle \rightarrow \text{for} (\langle \text{ForInit} \rangle_{\text{opt}} ; \langle \text{Expression} \rangle_{\text{opt}} ; \langle \text{ForUpdate} \rangle_{\text{opt}})$
 $\langle \text{StatementNoShortIf} \rangle$
95. $\langle \text{ForInit} \rangle \rightarrow \langle \text{StatementExpressionList} \rangle \mid \langle \text{LocalVariableDeclaration} \rangle$
96. $\langle \text{ForUpdate} \rangle \rightarrow \langle \text{StatementExpressionList} \rangle$
97. $\langle \text{StatementExpressionList} \rangle \rightarrow \langle \text{StatementExpression} \rangle \mid$
 $\langle \text{StatementExpressionList} \rangle , \langle \text{StatementExpression} \rangle$

Break, Continue, Return, Throw, Synchronized, and Try Statements

98. $\langle \text{BreakStatement} \rangle \rightarrow \text{break } \langle \text{Identifier} \rangle_{\text{opt}} ;$
99. $\langle \text{ContinueStatement} \rangle \rightarrow \text{continue } \langle \text{Identifier} \rangle_{\text{opt}} ;$
100. $\langle \text{ReturnStatement} \rangle \rightarrow \text{return } \langle \text{Expression} \rangle_{\text{opt}} ;$
101. $\langle \text{ThrowStatement} \rangle \rightarrow \text{throw } \langle \text{Expression} \rangle ;$
102. $\langle \text{SynchronizedStatement} \rangle \rightarrow \text{synchronized} (\langle \text{Expression} \rangle) \langle \text{Block} \rangle$
103. $\langle \text{TryStatement} \rangle \rightarrow \text{try } \langle \text{Block} \rangle \langle \text{Catches} \rangle \mid$
 $\text{try } \langle \text{Block} \rangle \langle \text{Catches} \rangle_{\text{opt}} \langle \text{Finally} \rangle$
104. $\langle \text{Catches} \rangle \rightarrow \langle \text{CatchClause} \rangle \mid \langle \text{Catches} \rangle \langle \text{CatchClause} \rangle$
105. $\langle \text{CatchClause} \rangle \rightarrow \text{catch} (\langle \text{FormalParameter} \rangle) \langle \text{Block} \rangle$

106. $\langle \text{Finally} \rangle \rightarrow \text{finally } \langle \text{Block} \rangle$

Creation and Access Expressions

107. $\langle \text{Primary} \rangle \rightarrow \langle \text{PrimaryNoNewArray} \rangle \mid \langle \text{ArrayCreationExpression} \rangle$

108. $\langle \text{PrimaryNoNewArray} \rangle \rightarrow \langle \text{Literal} \rangle \mid \text{this} \mid$
 $(\langle \text{Expression} \rangle) \mid \langle \text{ClassInstanceCreationExpression} \rangle \mid$
 $\langle \text{FieldAccess} \rangle \mid \langle \text{MethodInvocation} \rangle \mid$
 $\langle \text{ArrayAccess} \rangle$

109. $\langle \text{ClassInstanceCreationExpression} \rangle \rightarrow \text{new } \langle \text{ClassType} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}})$

110. $\langle \text{ArgumentList} \rangle \rightarrow \langle \text{Expression} \rangle \mid \langle \text{ArgumentList} \rangle , \langle \text{Expression} \rangle$

111. $\langle \text{ArrayCreationExpression} \rangle \rightarrow \text{new } \langle \text{PrimitiveType} \rangle \langle \text{DimExprs} \rangle \langle \text{Dims} \rangle_{\text{opt}} \mid$
 $\text{new } \langle \text{TypeName} \rangle \langle \text{DimExprs} \rangle \langle \text{Dims} \rangle_{\text{opt}}$

112. $\langle \text{DimExprs} \rangle \rightarrow \langle \text{DimExpr} \rangle \mid \langle \text{DimExprs} \rangle \langle \text{DimExpr} \rangle$

113. $\langle \text{DimExpr} \rangle \rightarrow [\langle \text{Expression} \rangle]$

114. $\langle \text{Dims} \rangle \rightarrow [] \mid \langle \text{Dims} \rangle []$

115. $\langle \text{FieldAccess} \rangle \rightarrow \langle \text{Primary} \rangle . \langle \text{Identifier} \rangle \mid \text{super} . \langle \text{Identifier} \rangle$

116. $\langle \text{MethodInvocation} \rangle \rightarrow \langle \text{MethodName} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}}) \mid$
 $\langle \text{Primary} \rangle . \langle \text{Identifier} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}}) \mid$
 $\text{super} . \langle \text{Identifier} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}})$

117. $\langle \text{ArrayAccess} \rangle \rightarrow \langle \text{ExpressionName} \rangle [\langle \text{Expression} \rangle] \mid$
 $\langle \text{PrimaryNoNewArray} \rangle [\langle \text{Expression} \rangle]$

Expressions

118. $\langle \text{Expression} \rangle \rightarrow \langle \text{AssignmentExpression} \rangle$

119. $\langle \text{ConstantExpression} \rangle \rightarrow \langle \text{Expression} \rangle$

Assignment Operators

120. $\langle \text{AssignmentExpression} \rangle \rightarrow \langle \text{ConditionalExpression} \rangle \mid \langle \text{Assignment} \rangle$

121. $\langle \text{Assignment} \rangle \rightarrow \langle \text{LeftHandSide} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle$

122. $\langle \text{LeftHandSide} \rangle \rightarrow \langle \text{ExpressionName} \rangle \mid \langle \text{FieldAccess} \rangle \mid \langle \text{ArrayAccess} \rangle$

123. $\langle \text{AssignmentOperator} \rangle \rightarrow = \mid * = \mid / = \mid \% = \mid + = \mid - = \mid < = \mid$
 $> = \mid > > = \mid \& = \mid \cdot \mid \mid =$

Postfix Expressions

124. $\langle \text{PostfixExpression} \rangle \rightarrow \langle \text{Primary} \rangle \mid \langle \text{ExpressionName} \rangle \mid$
 $\langle \text{PostIncrementExpression} \rangle \mid \langle \text{PostDecrementExpression} \rangle$

125. $\langle \text{PostIncrementExpression} \rangle \rightarrow \langle \text{PostfixExpression} \rangle ++$

126. $\langle \text{PostDecrementExpression} \rangle \rightarrow \langle \text{PostfixExpression} \rangle --$

Unary Operators

127. $\langle \text{UnaryExpression} \rangle \rightarrow \langle \text{PreIncrementExpression} \rangle \mid \langle \text{PreDecrementExpression} \rangle \mid$
 $+ \langle \text{UnaryExpression} \rangle \mid - \langle \text{UnaryExpression} \rangle \mid$
 $\langle \text{UnaryExpressionNotPlusMinus} \rangle$

128. $\langle \text{PreIncrementExpression} \rangle \rightarrow ++ \langle \text{UnaryExpression} \rangle$

129. $\langle \text{PreDecrementExpression} \rangle \rightarrow -- \langle \text{UnaryExpression} \rangle$

130. $\langle \text{UnaryExpressionNotPlusMinus} \rangle \rightarrow \langle \text{PostfixExpression} \rangle \mid \langle \text{UnaryExpression} \rangle \mid$
 $! \langle \text{UnaryExpression} \rangle \mid \langle \text{CastExpression} \rangle$

131. $\langle \text{CastExpression} \rangle \rightarrow (\langle \text{PrimitiveType} \rangle \langle \text{Dims} \rangle_{\text{opt}}) \langle \text{UnaryExpression} \rangle \mid$
 $(\langle \text{PrimitiveType} \rangle) \langle \text{UnaryExpressionNotPlusMinus} \rangle$

Operators

132. $\langle \text{MultiplicativeExpression} \rangle \rightarrow \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle * \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle / \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle \% \langle \text{UnaryExpression} \rangle$ [637]
133. $\langle \text{AdditiveExpression} \rangle \rightarrow \langle \text{MultiplicativeExpression} \rangle |$
 $\langle \text{AdditiveExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle |$
 $\langle \text{AdditiveExpression} \rangle - \langle \text{MultiplicativeExpression} \rangle$
134. $\langle \text{ShiftExpression} \rangle \rightarrow \langle \text{AdditiveExpression} \rangle |$
 $\langle \text{ShiftExpression} \rangle << \langle \text{AdditiveExpression} \rangle |$
 $\langle \text{ShiftExpression} \rangle >> \langle \text{AdditiveExpression} \rangle |$
 $\langle \text{ShiftExpression} \rangle >>> \langle \text{AdditiveExpression} \rangle$
135. $\langle \text{RelationalExpression} \rangle \rightarrow \langle \text{ShiftExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle < \langle \text{ShiftExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle > \langle \text{ShiftExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle < = \langle \text{ShiftExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle > = \langle \text{ShiftExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle \text{ instanceof } \langle \text{ReferenceType} \rangle$
136. $\langle \text{EqualityExpression} \rangle \rightarrow \langle \text{RelationalExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle == \langle \text{RelationalExpression} \rangle |$
 $\langle \text{RelationalExpression} \rangle != \langle \text{RelationalExpression} \rangle$
137. $\langle \text{AndExpression} \rangle \rightarrow \langle \text{EqualityExpression} \rangle | \langle \text{AndExpression} \rangle \& \langle \text{EqualityExpression} \rangle$
138. $\langle \text{ExclusiveOrExpression} \rangle \rightarrow \langle \text{EqualityExpression} \rangle |$
 $\langle \text{ExclusiveOrExpression} \rangle | \langle \text{AndExpression} \rangle$
139. $\langle \text{InclusiveOrExpression} \rangle \rightarrow \langle \text{ExclusiveOrExpression} \rangle |$
 $\langle \text{InclusiveOrExpression} \rangle | \langle \text{AndExpression} \rangle$
140. $\langle \text{ConditionalAndExpression} \rangle \rightarrow \langle \text{InclusiveOrExpression} \rangle |$
 $\langle \text{ConditionalAndExpression} \rangle \&\&$
 $\langle \text{InclusiveOrExpression} \rangle$
141. $\langle \text{ConditionalOrExpression} \rangle \rightarrow \langle \text{ConditionalAndExpression} \rangle |$
 $\langle \text{ConditionalOrExpression} \rangle ||$
 $\langle \text{ConditionalAndExpression} \rangle$
142. $\langle \text{ConditionalExpression} \rangle \rightarrow \langle \text{ConditionalOrExpression} \rangle |$
 $\langle \text{ConditionalOrExpression} \rangle ? \langle \text{Expression} \rangle :$
 $\langle \text{ConditionalExpression} \rangle$

Literals

143. $\langle \text{Literal} \rangle \rightarrow \langle \text{IntegerLiteral} \rangle | \langle \text{FloatingPointLiteral} \rangle | \langle \text{BooleanLiteral} \rangle |$
 $\langle \text{CharacterLiteral} \rangle | \langle \text{StringLiteral} \rangle | \langle \text{NullLiteral} \rangle$
144. $\langle \text{IntegerLiteral} \rangle \rightarrow \langle \text{DecimalIntegerLiteral} \rangle | \langle \text{HexIntegerLiteral} \rangle |$
 $\langle \text{OctalIntegerLiteral} \rangle$
145. $\langle \text{DecimalIntegerLiteral} \rangle \rightarrow \langle \text{DecimalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{opt}$
146. $\langle \text{HexIntegerLiteral} \rangle \rightarrow \langle \text{HexNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{opt}$
147. $\langle \text{HexIntegerLiteral} \rangle \rightarrow \langle \text{HexNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{opt}$
148. $\langle \text{OctalIntegerLiteral} \rangle \rightarrow \langle \text{OctalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{opt}$

149. $\langle \text{IntegerTypeSuffix} \rangle \rightarrow 1 \mid \text{L}$
150. $\langle \text{DecimalNumeral} \rangle \rightarrow 0 \mid \langle \text{NonZeroDigit} \rangle \langle \text{Digits} \rangle_{\text{opt}}$
151. $\langle \text{Digits} \rangle \rightarrow \langle \text{Digit} \rangle \mid \langle \text{Digits} \rangle \langle \text{Digit} \rangle$
152. $\langle \text{Digit} \rangle \rightarrow 0 \mid \langle \text{NonZeroDigit} \rangle$
153. $\langle \text{NonZeroDigit} \rangle \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
154. $\langle \text{HexNumeral} \rangle \rightarrow 0\text{x} \langle \text{HexDigit} \rangle \mid 0\text{X} \langle \text{HexDigit} \rangle \mid \langle \text{HexNumeral} \rangle \langle \text{HexDigit} \rangle$
155. $\langle \text{HexDigit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E}$
156. $\langle \text{OctalNumeral} \rangle \rightarrow 0 \langle \text{OctalDigit} \rangle \mid 0 \langle \text{OctalNumeral} \rangle \langle \text{OctalDigit} \rangle$
157. $\langle \text{OctalDigit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$
158. $\langle \text{FloatingPointLiteral} \rangle \rightarrow \langle \text{Digits} \rangle . \langle \text{Digits} \rangle_{\text{opt}} \langle \text{ExponentPart} \rangle_{\text{opt}} \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{\text{opt}} \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{\text{opt}} \langle \text{FloatTypeSuffix} \rangle$
159. $\langle \text{ExponentPart} \rangle \rightarrow \langle \text{ExponentIndicator} \rangle \langle \text{SignedInteger} \rangle$
160. $\langle \text{ExponentIndicator} \rangle \rightarrow \text{e} \mid \text{E}$
161. $\langle \text{SignedInteger} \rangle \rightarrow \langle \text{Sign} \rangle_{\text{opt}} \langle \text{Digits} \rangle$
162. $\langle \text{Sign} \rangle \rightarrow + \mid -$
163. $\langle \text{FloatTypeSuffix} \rangle \rightarrow \text{f} \mid \text{F} \mid \text{d} \mid \text{D}$
164. $\langle \text{BooleanLiteral} \rangle \rightarrow \text{true} \mid \text{false}$
165. $\langle \text{CharacterLiteral} \rangle \rightarrow ' \langle \text{InputCharacter} \rangle ' \mid ' \langle \text{EscapeCharacter} \rangle '$
166. $\langle \text{StringLiteral} \rangle \rightarrow " \langle \text{StringCharacters} \rangle_{\text{opt}} "$
167. $\langle \text{NullLiteral} \rangle \rightarrow \text{null}$

Identifier

168. $\langle \text{Identifier} \rangle \rightarrow \langle \text{IdentifierChars} \rangle$
169. $\langle \text{IdentifierChars} \rangle \rightarrow \langle \text{JavaLetter} \rangle \mid \langle \text{IdentifierChars} \rangle \langle \text{JavaLetterOrDigit} \rangle$

变量 $\langle \text{SingleCharacter} \rangle$ 、 $\langle \text{InputCharacter} \rangle$ 、 $\langle \text{EscapeSequence} \rangle$ 以及 $\langle \text{JavaLetter} \rangle$ 定义了可以用于输入、文字以及标识符的 16 - 位 Unicode 字符集的子集。

标识符是通过变量 $\langle \text{Identifier} \rangle$ 来定义的, 并且使用 Unicode 字母表的字符, 这样编程人员就可以使用他们自己的语言来书写源代码。标识符的第一个字符必须是字母、下划线或者是美元符号, 后边可以跟随任何个数的 Java 字母或者数字。Java 字母或者数字包含了可以使方法字符 `isJavaIdentifierPart` 返回 `true` 的 Unicode 字符。Java 关键字被保留起来, 并且不能被当作标识符来使用。

输入字符是 Unicode 字符, 不包括代表换行或者回车键的符号。 $\langle \text{SingleCharacter} \rangle$ 是输入字符, 但是不是 `'` 或者 `\`。换码序列包含一个 `\`, 后边跟随着 ASCII 符号来表示一个非图形字符。例如, `\n` 就是换行序列, 它代表着换行。有关 Java 编程语言的语法和语义的详细情况可以在 Gosling 等人编写的著作中找到 [2000]。

参考文献

- Ackermann, W. [1928], "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen*, 99, pp. 118-133.
- Aho, A. V. , and J. D. Ullman [1972], *The Theory of Parsing, Translation and Compilation*, Vol. I: *Parsing*, Prentice Hall, Englewood Cliffs, NJ.
- Aho, A. V. , and J. D. Ullman [1973], *The Theory of Parsing, Translation and Compilation*, Vol. II: *Compiling*, Prentice Hall, Englewood Cliffs, NJ.
- Aho, A. V. , J. E. Hopcroft, and J. D. Ullman [1974], *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
- Aho, A. V. , R. Sethi, and J. D. Ullman [1986], *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.
- Backus, J. W. [1959], "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. of the International Conference on Information Processing*, pp. 125-132.
- Bar-Hillel, Y. , M. Perles, and E. Shamir [1961], "On formal properties of simple phrasestructure grammars," *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, 14, pp. 143-177.
- Bavel, Z. [1983], *Introduction to the Theory of Automata*, Reston Publishing, Reston, VA.
- Blum, M. [1967], "A machine independent theory of the complexity of recursive functions," *J. ACM*, 14, pp. 322-336.
- Blum, M. , and R. Koch [1999], "Greibach normal form transformation, revisited," *Information and Computation*, 150, pp. 112-118.
- Bohrow, L. S. , and M. A. Arbib [1974], *Discrete Mathematics: Applied Algebra for Computer and Information Science*, Saunders, Philadelphia, PA.
- Bondy, J. A. , and U. S. R. Murty [1977], *Graph Theory with Applications*, Elsevier, New York.
- Brainerd, W. S. , and L. H. Landweber [1974], *Theory of Computation*, Wiley, New York.
- Brassard, G. , and P. Bratley [1996], *Fundamentals of Algorithms*, Prentice Hall, Englewood Cliffs, NJ.
- Busacker, R. G. , and T. L. Saaty [1965], *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York.
- Cantor, D. C. [1962], "On the ambiguity problems of Backus systems," *J. ACM*, 9, pp. 477-479.
- Cantor, G. [1947], *Contributions to the Foundations of the Theory of Transfinite Numbers* (reprint) , Dover, New York.
- Chomsky, N. [1956], "Three models for the description of languages," *IRE Trans. on Information Theory*, 2, pp. 113-124.
- Chomsky, N. [1959], "On certain formal properties of grammars," *Information and Control*, 2, pp. 137-167.
- Chomsky, N. [1962], "Context-free grammar and pushdown storage," *Quarterly Progress Report* 65, M. I. T. Research Laboratory in Electronics, pp. 187-194.
- Chomsky, N. , and G. A. Miller [1958], "Finite state languages," *Information and Control*, 1, pp. 91-112.
- Chomsky, N. , and M. P. Schutzenberger [1963], "The algebraic theory of context free languages," in *Computer Programming and Formal Systems*, North-Holland, Amsterdam, pp. 118-161.
- Christofides, N. [1975], "Worst case analysis of a new heuristic for the traveling salesman problem," *Research Report* 338, Management Sciences, Carnegie Mellon University, Pittsburgh, PA.

- Church, A. 1936, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, 58, pp. 345-363.
- Church, A. 1941, "The calculi of lambda-conversion," *Annals of Mathematics Studies*, 6, Princeton University Press, Princeton, NJ.
- Cobham, A. 1964, "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 24-30.
- Cook, S. A. 1971, "The complexity of theorem proving procedures," *Proc. of the Third Annual ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 151-158.
- Cook, S. A., and R. A. Reckhow 1973, "Time bounded random access machines," *Journal of Computer and System Science*, 7, pp. 354-375.
- Cormen T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. [2001], *Introduction to Algorithms*, McGraw-Hill, New York, NY.
- Davis, M. D. [1965], *The Undecidable*, Raven Press, Hewlett, NY.
- Davis, M. D., and E. J. Weyuker 1983, *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science*, Academic Press, New York.
- Denning, P. J., J. B. Dennis, and J. E. Qualitz 1978, *Machines, Languages, and Computation*, Prentice Hall, Englewood Cliffs, NJ.
- De Remer, F. L. 1969, "Generating parsers for BNF grammars," *Proc. of the 1969 Fall Joint Computer Conference*, AFIPS Press, Montvale, NJ, pp. 793-799.
- De Remer, F. L. [1971], "Simple LR(k) grammars," *Comm. ACM*, 14, pp. 453-460.
- Edmonds, J. 1965, "Paths, trees and flowers," *Canadian Journal of Mathematics*, 3, pp. 449-467.
- Engelfriet, J. 1992, "An elementary proof of double Greibach normal form," *Information Processing Letters*, 44, pp. 291-293.
- Evey, J. 1963, "Application of pushdown store machines," *Proc. of the 1963 Fall Joint Computer Science Conference*, AFIPS Press, pp. 215-217.
- Fischer, P. C. 1963, "On computability by certain classes of restricted Turing machines," *Proc. of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, pp. 23-32.
- Floyd, R. W. 1962, "On ambiguity in phrase structure languages," *Comm. ACM*, 5, pp. 526-534.
- Floyd, R. W. 1964, *New Proofs and Old Theorems in Logic and Formal Linguistics*, Computer Associates, Wakefield, MA.
- Foster, J. M. [1968], "A syntax improving program," *Computer J.*, 11, pp. 31-34.
- Fraenkel, A. A., Y. Bar-Hillel, and A. Levy [1984], *Foundations of Set Theory*, 2d ed., North-Holland, New York.
- Garey, M. R., and D. S. Johnson 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York.
- Ginsburg, S. 1966, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York.
- Ginsburg, S., and H. G. Rice 1962, "Two families of languages related to ALGOL," *J. ACM*, 9, pp. 350-371.
- Ginsburg, S., and G. F. Rose 1963a, "Some recursively unsolvable problems in ALGOLlike languages," *J. ACM*, 10, pp. 29-47.
- Ginsburg, S., and G. F. Rose 1963b, "Operations which preserve definability in languages," *J. ACM*, 10, pp. 175-195.
- Ginsburg, S., and J. S. Ullian 1966a, "Ambiguity in context-free languages," *J. ACM*, 13, pp. 62-89.
- Ginsburg, S., and J. S. Ullian 1966b, "Preservation of unambiguity and inherent ambiguity in context-free languages," *J. ACM*, 13, pp. 364-368.

- Gödel, K. [1931], "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I," *Monatshefte für Mathematik und Physik*, 38, pp. 173-198. (English translation in Davis [1965].)
- Gosling, J., B. Joy, G. Steele, and G. Bracha [2000], *The Java Language Specification*, 2d ed., Addison-Wesley, Boston, MA.
- Greibach, S. [1965], "A new normal form theorem for context-free phrase structure grammars," *J. ACM*, 12, pp. 42-52.
- Halmos, P. R. [1974], *Naive Set Theory*, Springer-Verlag, New York.
- Harrison, M. A. [1978], *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA.
- Hartmanis, J., and J. E. Hopcroft [1971], "An overview of the theory of computational complexity," *J. ACM*, 18, pp. 444-475.
- Hennie, F. C. [1977], *Introduction to Computability*, Addison-Wesley, Reading, MA.
- Hermes, H. [1965], *Enumerability, Decidability, Computability*, Academic Press, New York.
- Hochbaum, D. S., ed., 1997, *Approximation Algorithms for NP-Complete Problems*, PWS Publishing, Boston, MA.
- Hopcroft, J. E. [1971], "An $n \log n$ algorithm for minimizing the states in a finite automaton," in *The Theory of Machines and Computation*, ed. by Z. Kohavi, Academic Press, New York, pp. 189-196.
- Hopcroft, J. E., and J. D. Ullman [1979], *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA.
- Ibarra, O. H., and C. E. Kim [1975], "Fast approximation problems for the knapsack and sum of subsets problems," *J. ACM*, 22, no. 4, pp. 463-468.
- Jensen, K., and N. Wirth [1974], *Pascal: User Manual and Report*, 2d ed., Springer-Verlag, New York.
- Karp, R. M. [1972], "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, Plenum Press, New York, pp. 85-104.
- Karp, R. M. [1986], "Combinatorics, complexity and randomness," *Comm. ACM*, 29, no. 2, pp. 98-109.
- Kfoury, A. J., R. N. Moll, and M. A. Arbib [1982], *A Programming Approach to Computability*, Springer-Verlag, New York.
- Kleene, S. C. [1936], "General recursive functions of natural numbers," *Mathematische Annalen*, 112, pp. 727-742.
- Kleene, S. C. [1952], *Introduction to Metamathematics*, Van Nostrand, Princeton, NJ.
- Kleene, S. C. [1956], "Representation of events in nerve nets and finite automata," in *Automata Studies*, ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, pp. 3-42.
- Knuth, D. E. [1965], "On the translation of languages from left to right," *Information and Control*, 8, pp. 607-639.
- Knuth, D. E. [1968], *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA.
- Koch, R., and N. Blum [1997], "Greibach normal form transformation revisited," *Proceedings STACS 97, Lecture Notes in Computer Science 1200*, Springer-Verlag, New York, pp. 47-54.
- Korenjak, A. J. [1969], "A practical method for constructing LR(k) processors," *Comm. ACM*, 12, pp. 613-623.
- Kurki-Suonio, R. [1969], "Notes on top-down languages," *BIT*, 9, pp. 225-238.
- Kuroda, S. Y. [1964], "Classes of languages and linear-bounded automata," *Information and Control*, 7, pp. 207-223.
- Ladner, R. E. [1975], "On the structure of polynomial time reducibility," *Journal of the ACM*, 22, pp. 155-171.
- Landweber, P. S. [1963], "Three theorems of phrase structure grammars of type 1," *Information and Control*,

- 6, pp. 131-136.
- Levitin, A. [2003], *The Design and Analysis of Algorithms*, Addison-Wesley, Boston, MA.
- Lewis, H. R., and C. H. Papadimitriou [1981], *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, NJ.
- Lewis, P. M., II, and R. E. Stearns [1968], "Syntax directed transduction," *J. ACM*, 15, pp. 465-488.
- Markov, A. A. [1961], *Theory of Algorithms*, Israel Program for Scientific Translations, Jerusalem.
- McNaughton, R., and H. Yamada [1960], "Regular expressions and state graphs for automata," *IEEE Trans. on Electronic Computers*, 9, pp. 39-47.
- Mealy, G. H. [1955], "A method for synthesizing sequential circuits," *Bell System Technical Journal*, 34, pp. 1045-1079.
- Meyer, A. R., and L. J. Stockmeyer [1973], "The equivalence problem for regular expressions with squaring requires exponential space," *Proc. of the Thirteenth Annual IEEE Symposium on Switching and Automata Theory*, pp. 125-129.
- Minsky, M. L. [1967], *Computation: Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ.
- Moore, E. F. [1956], "Gedanken-experiments on sequential machines," in *Automata Studies*, ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, pp. 129-153.
- Myhill, J. [1957], "Finite automata and the representation of events," *WADD Technical Report 57-624*, Wright Patterson Air Force Base, OH, pp. 129-153.
- Myhill, J. [1960], "Linear bounded automata," *WADD Technical Note 60-165*, Wright Patterson Air Force Base, OH.
- Naur, P., ed. [1963], "Revised report on the algorithmic language ALGOL 60," *Comm. ACM*, 6, pp. 1-17.
- Nerode, A. [1958], "Linear automaton transformations," *Proc. AMS*, 9, pp. 541-544.
- Oettinger, A. G. [1961], "Automatic syntax analysis and the pushdown store," *Proc. on Symposia on Applied Mathematics*, 12, American Mathematical Society, Providence, RI, pp. 104-129.
- Ogden, W. G. [1968], "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory*, 2, pp. 191-194.
- Ore, O. [1963], *Graphs and Their Uses*, Random House, New York.
- Papadimitriou, C. H. [1994], *Computational Complexity*, Addison-Wesley, Reading, MA.
- Papadimitriou, C. H., and K. Steiglitz [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, NJ.
- Parikh, R. J. [1966], "On context-free languages," *J. ACM*, 13, pp. 570-581.
- Péter, R. [1967], *Recursive Functions*, Academic Press, New York.
- Post, E. L. [1936], "Finite combinatory processes—formulation I," *Journal of Symbolic Logic*, 1, pp. 103-105.
- Post, E. L. [1946], "A variant of a recursively unsolvable problem," *Bulletin of the American Mathematical Society*, 52, pp. 264-268.
- Post, E. L. [1947], "Recursive unsolvability of a problem of Thue," *Journal of Symbolic Logic*, 12, pp. 1-11.
- Pratt, V. [1975], "Everyprime has a succinct certificate," *SIAM Journal of Computation*, 4, pp. 214-220.
- Rabin, M. O., and D. Scott [1959], "Finite automata and their decision problems," *IBM J. Res.*, 3, pp. 115-125.
- Rice, H. G. [1953], "Classes of recursively enumerable sets and their decision problems," *Trans. of the American Mathematical Society*, 89, pp. 25-29.
- Rice, H. G. [1956], "On completely recursively enumerable classes and their key arrays," *Journal of Symbolic Logic*, 21, pp. 304-341.
- Rogers, H., Jr. [1967], *Theory of Recursive Functions and Effective Computation*, McGrawHill, New York.

- Rosenkrantz, D. J. , and R. E. Stearns [1970] , " Properties of deterministic top-down grammars , " *Information and Control* , 17 , pp. 226-256 .
- Sahni, A. [1975] , " Approximate algorithms for the 0/1 knapsack problem , " *J. ACM* , 22 , no. 1 , pp. 115-124 .
- Salomaa, A. [1973] , *Formal Languages* , Academic Press , New York .
- Salomaa, S. [1966] , " Two complete axiom systems for the algebra of regular events , " *J. ACM* , 13 , pp. 156-199 .
- Savitch, W. J. [1970] , " Relationships between nondeterministic and deterministic tape complexities , " *J. Computer and Systems Sciences* , 4 , no. 2 , pp. 177-192 .
- Scheinberg, S. [1960] , " Note on the Boolean properties of context-free languages , " *Information and Control* , 3 , pp. 372-375 .
- Schutzenberger, M. P. [1963] , " On context-free languages and pushdown automata , " *Information and Control* , 6 , pp. 246-264 .
- Sheperdson, J. C. [1959] , " The reduction of two-way automata to one-way automata , " *IBM J. Res.* , 3 , pp. 198-200 .
- Sheperdson, J. C. , and H. E. Sturgis [1963] , " Computability of recursive functions , " *J. ACM* , 10 , pp. 217-255 .
- Soisalon-Soininen, E. , and E. Ukkonen [1979] , " A method for transforming grammars into $LL(k)$ form " *Acta Informatica* , 12 , pp. 339-369 .
- Stearns, R. E. [1971] , " Deterministic top-down parsing , " *Proc. of the Fifth Annual Princeton Conference of Information , Sciences and Systems* , pp. 182-188 .
- Stoll, R. [1963] , *Set Theory and Logic* , W. H. Freeman , San , Francisco , CA .
- Thue, A. [1914] , " Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln , " *Skrifter utgit av Videnskapselskapet i Kristiana* , I. , Matematisknatur-videnskabelig klasse , 10 .
- Turing, A. M. [1936] , " On computable numbers with an application to the Entscheidungsproblem , " *Proc. of the London Mathematical Society* , 2 , no. 42 , pp. 230-265 ; no. 43 , pp. 544-546 .
- von Neumann, J. [1945] , *First Draft of a Report on EDVAC* , Moore School of Electrical Engineering , University of Pennsylvania , Philadelphia , PA . Reprinted in : Stern, N. [1981] , *From Eniac to Univac* , Digital Press , Bedford , MA .
- Wand, M. [1980] , *Induction, Recursion and Programming* , North-Holland , New York .
- Wilson, R. J. [1985] , *Introduction to Graph Theory* , 3d ed. , American Elsevier , New , York .
- Wood, D. [1969] , " The theory of left factored languages , " *Computer Journal* , 12 , pp. 349-356 .
- Younger, D. [1967] , " Recognition and parsing of context-free languages in time n^3 , " *Information and Control* , 10 , no. 2 , pp. 189-208 .

索引

索引中页码为英文原书页码,与书中页边标注的页码一致。

A

Abnormal termination 异常终止, 257
Abstract machine 抽象机, 147
Acceptance
 by deterministic finite automaton 被确定型有限自动机接收, 147 - 148
 by empty stack 被空栈方式接收, 230
 by entering 被进入方式接收, 263, 289
 by final state 被终结状态方式接收, 229, 260
 by halting 被停机方式接收, 262 - 263
 by nondeterministic finite automaton 被非确定型有限自动机接收, 161
 by nondeterministic Turing machine 被非确定型图灵机接收, 274
 by pushdown automaton 被下推自动机接收, 224, 229 - 230
 by Turing machine 被图灵机接收, 260
Accepted string 被接收的串, 148, 224
Accepting state 接收状态, 146 - 147
Ackermann's function 阿克曼函数, 411 - 413
Acyclic graph 无环图, 33
Adjacency relation 邻接关系, 32
AE, 205, 556, 585
 nonregularity of (AE 的) 非正则性, 205
ALGOL, 1, 94, 553
Algorithm 算法, 343 - 344
Alphabet 字母表, 42 - 43, 147, 163
 input 输入 (字母表), 222, 256
 stack 栈 (字母表), 222
 tape 带 (字母表), 256
Ambiguity 二义性, 91 - 93
 inherent 固有的 (二义性), 92
Ancestor 祖先, 34 - 35
Approximation algorithm 近似算法, 519 - 521
Approximation schema 近似方案, 523 - 526
 fully polynomial 完全多项式的 (逼近格式), 526
Arithmetization 算术化, 416
ASCII character set ASCII 字符集, 21 - 22
Associativity 相关性, 44 - 45
Atomic pushdown automaton 原子下推自动机, 227
Atomic Turing machine 原子图灵机, 290

B

Backus-Naur Form 巴克斯-瑙尔范式, 94, 553, 631
Barber's paradox 理发师悖论, 21 - 22
Big oh 大 O 渐进标记法, 436 - 438
Big theta 大 θ 渐进标记法, 438
Bin Packing Problem 装箱问题, 516
Binary relation 二元关系, 11 - 12
Binary tree 二叉树, 35
Blank Tape Problem 空带问题, 366 - 368
BNF BNF 范式, 94, 553, 631
Boolean variable 布尔变量, 481
Bottom-up parser 自底向上分析, 555, 561 - 567
 depth-first 深度优先的 (自底向上分析) 563 - 567
 LR(0) LR(0) 的 (自底向上分析), 599 - 601, 604
 LR(1) LR(1) 的 (自底向上分析), 618
Bounded operators 有界操作符, 398 - 404
Bounded sum 有界和, 398
Breadth-first bottom-parser 宽度优先自底向上分析器, 563 - 567
Breadth-first top-down parser 宽度优先自顶向下分析器, 557 - 561

C

Cardinality 基数, 16 - 21
Cartesian product 笛卡儿集, 11 - 12
Chain 链, 114
Chain rule 链规则, 113
 elimination of (链规则的) 消除, 113 - 116
Characteristic function 特征函数, 298 - 299
Child 子节点, 33
Chomsky hierarchy 乔姆斯基层次, 64, 338 - 339
Chomsky normal form 乔姆斯基范式, 121 - 124, 239 - 240
Church-Turing Thesis 丘奇-图灵论题, 2, 253, 344, 352 - 354, 421 - 423
Clause 语句, 482
Closure properties 封闭特性
 of context-free languages 上下文无关语言的 (封闭特性), 243 - 246
 of countable sets 可数集合的 (封闭特性), 18 - 19

of regular languages 正则语言的 (封闭特性), 200 - 203
 Co-NP, 531
 Compatible transitions 兼容转移, 225
 Compilation 编译, 567
 Complement 补, 9
 acceptance of (补) 的接收, 158
 Complete binary tree 完全二叉树, 40
 Complete item 完全项, 602
 Complexity 复杂性, 433 - 436
 nondeterministic 非确定型的 (复杂性), 466 - 468
 space complexity 空间 (复杂性), 532 - 535
 time complexity 时间 (复杂性), 442 - 446
 Composition of functions 函数的合成, 308 - 309
 Computable function 可计算函数, 7, 296, 353
 Church-Turing Thesis for 用于可计算函数的丘奇—图灵论
 题, 353 - 354, 421 - 423
 Concatenation 连接
 of languages 语言的 (连接), 47
 of strings 串的 (连接), 43 - 44
 Conjunctive normal form 合取范式, 482
 Context 上下文, 69
 Context-free grammar 上下文无关文法, 68 - 69
 ambiguous 二义性的 (上下文无关文法), 91, 384
 equivalent 等价的 (上下文无关文法), 79
 for Java 用于 Java 的 (上下文无关文法), 94 - 97,
 631 - 639
 language of (上下文无关文法) 的语言, 70
 left-linear 左线性的, 220
 left-regular 左正则的, 219 - 220
 right-linear 右线性的, 102, 219
 Context-free grammar 上下文无关文法
 undecidable problems of (上下文无关文法) 的不可判定
 问题, 382 - 386
 Context-free language 上下文无关语言, 70
 acceptance by pushdown automaton 被下推自动机接收
 (的上下文无关语言), 232 - 239
 closure properties of (上下文无关语言) 封闭性质,
 243 - 246
 examples of (上下文无关语言的) 例子, 76 - 81
 inherently ambiguous (上下文无关语言的) 泵引理, 92
 pumping lemma for 用于 (上下文无关语言的) 泵引理,
 239 - 242
 Context-sensitive grammar 上下文有关文法, 332 - 334
 Context-sensitive language 上下文有关语言, 333
 Context-sensitive Turing machine 上下文有关的图灵机, 290
 Cook's Theorem 库克定理, 485
 Countable set 可数集, 7, 17 - 19
 Countably infinite set 可数无限集, 17 - 19
 Course-of-values recursion 计值过程递归, 409
 Cycle 循环, 33

Cyclic graph 循环图, 33
 CYK algorithm CYK 算法, 124 - 128

D

Dead end 死端, 558
 Decidable 可判定的
 language (可确定的) 语言, 260
 in polynomial space 多项式空间可判定的, 540
 in polynomial time 多项式时间可判定的, 468
 problem 可判定的问题, 343 - 344
 Decision problem 判定问题, 343 - 346
 Bin Packing Problem 装箱问题, 516
 Blank Tape Problem 空带问题, 366 - 368
 Church-Turing Thesis for 用于 (判定问题的) 丘奇—图
 灵论题, 353
 Halting Problem 停机问题, 357, 362 - 365
 Hamiltonian Circuit Problem 哈密尔顿通路问题,
 473 - 477, 503 - 509
 Hitting Set Problem 命中集问题, 515 - 516
 intractable 难处理的 (确定问题), 431, 465, 548 - 550
 Knapsack Problem 背包问题, 518
 NP-complete NP-完全, 480
 Partition Problem 分割问题, 513 - 515
 Post Correspondence Problem 波斯特对应问题, 377 - 382
 reduction of (判定问题的) 归约, 348 - 352
 representation of (判定问题的) 陈述, 344 - 346,
 469 - 471
 Satisfiability Problem (判定问题的) 可满足性问题,
 472, 481 - 483
 Subset-Sum Problem (判定问题的) 子集—和问题,
 473, 509 - 513
 3-Satisfiability Problem (判定问题的) 3 - 可满足性问
 题, 498 - 500
 Traveling Salesman Problem 旅行推销员问题, 517 - 518
 undecidable 不可判定的 (判定问题), 361
 Vertex Cover Problem (判定问题的) 顶点覆盖问题,
 500 - 503, 527
 Word Problem (判定问题的) 词组问题, 373 - 376
 DeMorgan's Laws 德摩根定理, 9 - 10
 Denumerable set 可数集, 17 - 19
 Depth of a node 节点深度, 34
 Derivable string 推导串, 69, 326
 Derivation 推导, 66 - 67, 69
 directly left recursive 直接左递归 (推导), 129
 leftmost 最左 (推导), 71, 89 - 91
 length of (推导的) 长度, 69
 recursive 递归 (推导), 71
 rightmost 最右 (推导), 71
 Derivation tree 推导树, 71 - 74

Descendant 后代, 34 - 35

Deterministic finite automaton (DFA) 确定型有限自动机, 2 - 3, 147

examples (确定型的有限自动机的) 例子, 150 - 159

extended transition function (确定型的有限自动机的) 扩展转换函数, 151

incompletely specified 不完全指定的 (确定型的有限自动机), 158

language of (确定型的有限自动机的) 语言, 148

minimization (确定型的有限自动机的) 最小化, 178 - 183

state diagram of (确定型的有限自动机的) 状态图, 146 - 147, 151 - 153

transition table (确定型的有限自动机的) 转换表, 150

Deterministic LR(0) machine 确定型 LR(0) 机, 603

Deterministic pushdown automaton 确定型的下推自动机, 225

DFA, 见确定型有限自动机

Diagonalization 对角线化, 7, 19 - 23

Difference of sets 集合的差, 9

Direct left recursion, removal of 直接左递归, 移除, 129 - 131

Directed graph 有向图, 32 - 33, 347

Disjoint set 不交的集合, 9

Distinguishable state 可区分的状态, 178

Distinguished element 可区分的元素, 32

Domain 定义域, 12

Dynamic programming 动态规划, 125

E

Effective procedure 有效过程, 253, 344

Empty set 空集, 8

Empty stack (acceptance by) 空栈 (由……接收), 230

Enumeration, by Turing machine 枚举, 通过图灵机, 282 - 288

Equivalence class 等价类, 15

Equivalence relation 等价关系, 14 - 16

right-invariant 右不变式 (的等价关系), 212

Equivalent

grammars (等价的) 文法, 79

machines (等价的) 机器, 158

regular expressions (等价的) 正则表达式, 53

states (等价的) 状态, 178

Essentially noncontracting grammar 本质上非收缩性文法, 110

Expanding a node 扩展节点, 558

Exponential growth 指数级增长, 441

Expression graph 表达式图, 193 - 196

Extended pushdown automaton 扩展的下推自动机, 225

Extended transition function 扩展转换函数, 151

F

Factorial 阶的, 392 - 393

Fibonacci numbers 斐波纳契数列, 407

Final state 终结状态, 147, 259

acceptance by 被 (终结状态) 接收, 229, 260

Finite automaton 有限自动机

Deterministic finite automaton 确定性有限自动机

Finite-state machine 有限状态机

Nondeterministic finite automaton 非确定型的有限自动机

Finite-state machine 有限状态机, 145 - 147

FIRST_k set FIRST_k 集合, 576

construction of (集合的) 构造, 580 - 583

Fixed point 稳定点, 20

FOLLOW_k set FOLLOW_k 集合, 577

construction of (集合的) 构造, 583 - 585

Frontier (of a tree) (树的) 周边, 35

Fully space constructible 完全空间可构造的, 538

Function 函数, 14

Ackermann's (函数), 411 - 413

characteristic 特征 (函数), 298 - 299

composition of (函数) 合成, 308 - 309

computable 可计算 (函数), 7

computation of 计算 (函数), 295 - 298

identity 恒等式 (函数), 301

input transition 输入状态转换 (函数), 170

macro-computable 宏可计算 (函数), 430

μ -recursive μ -递归 (函数), 414

n -variable n -变量 (函数), 12

number-theoretic 数论 (函数), 299

one-to-one 1 对 1 (函数), 13 - 14

onto 在上 (函数), 13 - 14

partial 部分 (函数), 13

polynomially bounded 有界多项式 (函数), 440 - 441

primitive recursive 原始递归 (函数), 389 - 391

projection 投影 (函数), 300, 390

rates of growth 增长率 (函数), 436 - 441

total 全 (函数), 13

transition 状态转换 (函数), 147, 163, 166, 222

Turing computable 图灵机可计算 (函数), 296

uncomputable 不可计算 (函数), 312 - 313

G

Gödel numbering 歌德尔编号方式, 406

Grammar 文法。见 Context-free grammar 上下文无关文法, Regular grammar 正则文法 context-sensitive 上下文有关 (文法), 332 - 334

essentially noncontracting 本质上非收缩性的 (文法), 110

graph of (文法) 图, 556

language of (文法) 语言, 70

linear 线性 (文法), 250

LL(k) LL(k) 文法, 571, 589–591

LR(0) LR(0) 文法, 598, 609

LR(1) LR(1) 文法, 612–618

noncontracting 非收缩性 (文法), 107

phrase-structure 短语结构 (文法), 325–326

regular 正则 (文法), 81–83, 196

right-linear 右线性 (文法), 102, 219

strong LL(k) 强 LL(k) (文法), 579–580

unrestricted 不受限 (文法), 254, 325–332

Graph 图

acyclic 无环 (图), 33

cyclic 循环 (图), 33, 358

directed 有向 (图), 32–34, 347

expression 表达 (图), 193–196

of a grammar 文法 (图), 556

Graph of a grammar 文法图, 556

Greibach normal form 格立巴赫范式, 131–138, 232–233

Grep (Unix 工具程序), 55–58

H

Halting 停机, 257

acceptance by 被停机方式接收, 262–263

Halting Problem 停机问题, 357, 362–365

Hamiltonian Circuit Problem 哈密尔顿回路问题, 473–477, 503–509

Hard for a class 对于类是困难的, 479

Hitting Set Problem 命中集问题, 515–516

Home position 中心位置, 314

Homomorphic image 同态象, 219

Homomorphism 同态, 219, 257

I

Identity function 恒等函数, 301

Implicit tree 隐含树, 557

In-degree of a node 点的入度, 32

Indistinguishable 不可区分的

state (不可区分的) 状态, 178

string (不可区分的) 串, 211–212

Induction 归纳. 见 Mathematical induction 数学归纳

Infinite set 无穷集, 17

Infix notation 中缀表示法, 205

Inherent ambiguity 固有二义性, 92

Input alphabet 输入字母表, 222

Input transition function 输入状态转换函数, 170

Intersection of sets 集合的交集, 9

Intractable decision problem 难解问题 431, 465, 548–550

Invariance 不变性, right 右, 212

Inverse homomorphic image 反同态象, 251

Item 项

complete 完全 (项), 602

LL(0) LL(0) 项, 602

LR(1) LR(1) 项, 614

J

Java, 94–97, 631–639

K

Kleene star operation 克林星号操作, 47–48

Kleene's Theorem 克林定理, 196

Knapsack Problem 背包问题, 518

approximation algorithm for (背包问题的) 逼近算法, 524–526

L

L_{REG} , 545

L_{SAT} , 483

λ -closure λ -闭包, 170

λ -rule λ -规则, 69

elimination of (λ -规则) 消除, 106–113

λ -transition λ -状态转换, 165–166

Language 语言, 41–43, 326

context-free 上下文无关 (语言), 70

context-sensitive 上下文有关 (语言), 332

finite specification of (语言) 的有限规格说明, 45–49

of finite-state machine 有限状态机 (语言), 63, 148

inherently ambiguous (语言) 的固有二义性, 92

of nondeterministic finite automaton 非确定型的有限自动机 (语言), 163

nonregular 非正则 (语言), 203–204

of phrase-structure grammar 短语构造文法 (语言), 326

polynomial 多项式 (语言), 468

of pushdown automaton 下推自动机 (语言), 224

recognition 识别 (语言), 260

recursive 递归 (语言), 260

recursive definition of (语言) 的递归定义, 46–47

recursively enumerable 递归可枚举 (语言), 260

regular 正则 (语言), 49, 82, 200

of Turing machine 图灵机 (语言), 260

Language acceptor, Turing machine as 语言接收器, 图灵机作为, 259–262

Language enumerator, Turing machine as 语言入口调查, 图灵机作为, 282 - 288

LBA. 见线性有界自动机

Leaf 叶, 34

Left factoring 提取左因子, 576

Left-linear context-free grammar 左线性上下文无关文法, 220

Left-recursive rule 左递归规则, 129

Left-regular context-free grammar 左正则上下文无关文法, 219 - 220

Leftmost derivation 最左推导, 71, 89 - 91
ambiguity and 无二义性 (的最左推导), 91 - 93

Lexical analysis 词法分析, 553, 567

Lexicographical ordering 字典顺序, 286

L'Hospital's Rule 洛必达法则, 439 - 440

Linear-bounded automaton (LBA) 线性有界自动机, 334 - 338

Linear grammar 线性文法, 250

Linear speedup 线性加速, 448 - 451

Literal 文字的, 482

LL(k) grammar LL(k) 文法, 571, 589 - 591
strong 强 (LL(k) 文法), 579 - 580

Lookahead 预读
set (预读) 集, 572, 575, 589
string (预读) 串, 572

Lower-order terms 低阶项, 436

LR(0)
context 上下文, 595 - 596
grammar 文法, 598, 609
item 项, 602
machine 机, 602 - 603, 606 - 610
parser 分析器, 599 - 601, 604

LR(1)
context 上下文, 613
grammar 文法, 612 - 618
item 项, 614
machine 机, 614 - 617
parser 分析器, 618

M

Machine configuration 机器格局
of deterministic finite automaton 确定性有限自动机的 (机器格局), 149
of pushdown automaton 下推自动机的 (机器格局), 224
of Turing machine 图灵机的 (机器格局), 256 - 258

Macro 宏, 302 - 305

Macro-computable function 宏可计算函数, 430

Mathematical induction 数学归纳, 27 - 32

simple 简单的 (数学归纳), 30

strong 强 (数学归纳), 30

Microsoft Word 微软的文字处理软件 Word, 58

Minimal common ancestor 最近公共祖先, 34 - 35

Minimalization 最低程度, 400
bounded 受限的最低程度, 401
unbounded 不受限的最低程度, 413

Monotonic rule 单调规则, 333

Moore machine 摩尔机, 156
 μ -recursive function μ -递归函数, 414
Turing computability of (μ -递归函数的) 图灵可计算性, 414 - 415

Multitape Turing machine 多带图灵机, 268 - 274
time complexity of (多带图灵机) 的时间复杂性, 447 - 448

Multitrack Turing machine 多道图灵机, 263 - 265
time complexity of (多道图灵机) 的时间复杂性, 446

Myhill-Nerode Theorem Myhill-Nerode 定理, 211 - 217

N

n -ary relation n 元关系, 12

n -variable function n -变量函数, 12

Natural language 自然语言, 1, 5

Natural numbers 自然数, 8
recursive definition of (自然数) 的递归定义, 24

NFA. 见 Nondeterministic finite automaton 非确定型的有限自动机

NFA- λ , 165 - 166

Node 节点, 32

Noncontracting derivation 非收缩性的推导, 333

Noncontracting grammar 非收缩性的文法, 107

Nondeterminism, removing 非确定性, 移除, 170 - 178

Nondeterministic complexity 非确定型的复杂性, 442 - 446

Nondeterministic finite automaton (NFA) 非确定型有限自动机, 159 - 163
acceptance by 被 (非确定型有限自动机) 接收, 161
examples (非确定型有限自动机) 的例子, 164 - 165
input transition function (非确定型有限自动机) 的输入状态转换函数, 169
 λ -transition (非确定型有限自动机) 的 λ -状态转移, 165 - 166
language of (非确定型有限自动机) 的语言, 163

Nondeterministic LR(0) machine 非确定型 LR(0) 机, 602 - 603

Nondeterministic LR(1) machine 非确定型 LR(1) 机, 616

Nondeterministic polynomial time 非确定的多项式时间, 469

Nondeterministic Turing machine 非确定型图灵机, 274 - 282

- Nonregular language 非正则语言, 203 - 205
- Nonterminal symbol 非终结符号, 65, 69
- Normal form 范式, 103
- Chomsky 乔姆斯基 (范式), 121 - 124, 239 - 240
- conjunctive 合取 (范式), 482
- Greibach 格立巴赫 (范式), 131 - 138, 232 - 235
- 3-conjunctive 3 - 合取 (范式), 498
- \mathcal{NP} , 431, 469
- \mathcal{NPC} , 492
- NP-complete problem NP-完全问题, 480
- Bin Packing Problem 装箱问题, 516
- Hamiltonian Circuit Problem 哈密尔顿回路问题, 473 - 477, 503 - 509
- Hitting Set Problem 命中集问题, 515 - 516
- Knapsack Problem 背包问题, 518
- Partition Problem 分割问题, 513 - 515
- Satisfiability Problem 可满足性问题, 473, 481 - 483
- Subset-Sum Problem 子集和问题, 473, 509 - 513
- 3-Satisfiability Problem 3 - 可满足性问题, 498 - 500
- Traveling Salesman Problem 旅行邮递员问题, 517 - 518
- Vertex Cover Problem 节点覆盖问题, 500 - 503, 527
- NP-hard problem NP-完全问题, 480
- \mathcal{NPJ} , 530
- \mathcal{NP} -Space \mathcal{NP} -空间, 540
- Null 空
- path (空) 路径, 33
- rule (空) 规则, 69
- string (空) 串, 42
- Nullable variable 可空变量, 107
- Number-theoretic function 数论函数, 299
- Numeric computation 数值计算, 299 - 301
- One-to-one function 1 对 1 函数, 13 - 14
- Onto function 满函数, 13 - 14
- Operator, bounded 操作符, 有界的, 398 - 404
- Optimization problem 最优化问题, 517 - 518
- Order of a function 函数的阶, 436
- Ordered n -tuple 有序 n 元组, 12
- Ordered tree 有序树, 33 - 34
- Out-degree of a node 节点的出度, 32
- Output tape 输出带, 282
- \mathcal{P} , 431, 468
- \mathcal{P} -Space \mathcal{P} -空间, 540
- completeness (\mathcal{P} -空间的) 完全性, 544 - 545
- Palindrome 回文, 60, 77 - 78, 204, 226
- Parsing 分析, 553, 567 - 568
- bottom-up parser 自底向上分析器, 555, 561 - 567
- breadth-first bottom-parser 宽度优先自底向上分析器, 553 - 567
- breadth-first top-down parser 宽度优先自顶向下分析器, 557 - 561
- CYK algorithm CYK 算法 (分析), 124 - 128
- deterministic 确定性的 (分析), 554, 571
- $LL(k)$ $LL(k)$ (分析), 591
- $LR(0)$ $LR(0)$ (分析), 599 - 601, 604
- $LR(1)$ $LR(1)$ (分析), 618
- strong $LL(k)$ 强 $LL(k)$ (分析), 587 - 588
- top-down 自顶向下 (分析), 555
- Partial function 部分函数, 13
- Partition 分割, 9
- Partition Problem 分割问题, 513 - 515
- Path 路径, 32 - 33
- null 空 (路径), 33
- PDA. 见 Pushdown automaton 下推自动机
- Phrase-structure grammar 短语 - 结构文法, 325 - 326
- Pigeonhole principle 鸽巢原理, 206
- Polynomial with integral coefficients 带积分系数的多项式, 438
- Polynomial language 多项式语言, 468
- Polynomially bounded function 有界多项式函数, 440 - 441
- Post Correspondence Problem 波斯特对应问题 377 - 382
- Post correspondence system 波斯特对应系统, 377
- Power set 幂集, 9, 20
- Prefix 前缀, 45
- terminal prefix 终结符前缀, 129, 557
- viable 活 (前缀), 599
- Primitive recursion 原始递归, 390 - 391
- Primitive recursive function 原始递归函数, 389 - 391, 410 - 413
- basic 基本的 (原始递归函数), 389 - 390
- examples of (原始递归函数) 的例子, 391 - 398
- Turing computability of 原始递归函数的图灵机可计算性, 393 - 394
- Primitive recursive predicate 原始递归谓词, 395
- Problem reduction 问题归约, 348 - 352, 365 - 367
- and NP-completeness 和 NP-完全, 497 - 498, 513 - 514
- polynomial-time 多项式时间 (的问题归约), 477
- and undecidability (问题归约) 和不可判定性, 365 - 368
- Projection function 投影函数, 300, 390
- Proof by contradiction 反证法, 19 - 23
- Proper subset 真子集, 9
- Proper subtraction 真减, 39, 395
- Pseudo-polynomial problem 伪多项式问题, 471
- Pumping lemma 泵引理
- for context-free languages 用于上下文无关语言的 (泵引

理), 239 - 242
 for regular languages 用于正则语言的 (泵引理), 205 - 209
 Pushdown automaton (PDA) 下推自动机, 2 - 3, 221 - 222
 acceptance 接收, 224
 acceptance by empty stack 被空栈方式接收, 230
 acceptance by final state 被终结状态接收, 229
 atomic 原子的 (下推自动机), 227
 context-free language and 上下文无关语言和 (下推自动机), 232 - 239
 deterministic 确定型的 (下推自动机), 225
 extended 扩展的 (下推自动机), 228
 language of (下推自动机) 的语言, 230
 stack alphabet (下推自动机的) 栈字母表, 222
 state diagram (下推自动机的) 状态图, 222 - 223
 variations (下推自动机的) 变量, 227 - 232
R
 Random access machine 随即访问机, 323
 Range 值域, 12
 Rates of growth 增长率, 436 - 441
 Reachable variable 可达变量, 119
 Recognition of language 语言识别, 260
 Recursion 递归
 course-of-values 求值过程, 409
 primitive 原语, 390 - 391, 413 - 414
 removal of direct left recursion 直接左递归的消除, 129 - 131
 simultaneous 同步 (递归), 427
 Recursive definition 递归定义, 23 - 27, 45 - 46
 Recursive language 递归语言, 260, 346
 Recursive variable 递归变量, 71, 390
 Recursively enumerable language 递归可枚举语言, 260
 Reduction 归约, 555 - 556, 561. 见 Problem reduction 问题归约
 Regular expression 正则表达式, 42, 50
 defining pattern with 用 (正则表达式) 定义模式, 54 - 58
 equivalent 等价的 (正则表达式), 53
 examples (正则表达式的) 例子, 51 - 53
 with squaring 用扯平测长法, 548 - 550
 Regular grammar 正则文法, 81 - 83, 196
 finite automaton and, 有限自动机, 196 - 200
 Regular language 正则语言, 49, 82, 200
 acceptance by finite automaton, 被有限自动机接收, 191 - 193
 closure properties of (正则语言的) 封闭性质, 200 - 203
 decision procedures for 用于 (正则语言的) 判定过程,

209 - 210
 generation by regular grammar 由正则文法生成, 198 - 199
 pumping lemma for 用于 (正则语言) 的泵引理, 205 - 209
 Regular set 正则集, 49 - 50
 finite automaton and 有限自动机和 (正则集), 191 - 193
 Relation 关系
 adjacency 邻接 (关系), 32
 binary 二元 (关系), 11 - 12
 characteristic function of (关系) 的特征函数, 299
 equivalence 等价 (关系), 14 - 16
 n -ary n 元关系, 12
 Turing computable 图灵机可计算的关系, 299
 Reversal of a string 串的逆, 45
 Rice's Theorem 莱斯定理, 371 - 373
 Right invariant equivalence relation 右不变等价关系, 212
 Right-linear grammar 右线性文法, 102, 219
 Rightmost derivation 最右推导, 71
 Root 根, 33
 Rule 规则, 65, 326
 chain rule 链规则, 113
 context-free 上下文无关 (规则), 65 - 66, 69
 λ -rule λ -规则, 69
 left-recursive 左递归 (规则), 70, 129
 monotonic 单调 (规则), 333
 null 空 (规则), 69
 of phrase-structure grammar 短语构造文法 (规则), 326
 recursive 递归 (规则), 67 - 68, 70
 regular 正则 (规则), 81
 right recursive 右递归 (规则), 70, 130
 of semi-Thue system 半图厄系统规则, 373
 of unrestricted grammar 无限制文法规则, 326
 Russell's paradox 罗素悖论, 21 - 23

S

Satisfiability Problem 可满足性问题, 472, 481 - 483
 NP-completeness of (可满足性问题) 的 NP-完全, 483 - 492
 Savitch's Theorem 萨维奇定理, 542
 Schröder-Bernstein Theorem 定理, 16 - 17
 Search tree 搜索树, 558 - 561
 Self-reference 自引用, 21 - 23, 363 - 364
 Semi-Thue system 半图厄系统, 373 - 376
 Sentence 句子, 65 - 68, 70
 Sentential form 句型, 70
 terminal prefix of 句型的终结符前缀, 129
 Set 集合, 8 - 11
 cardinality of (集合) 的势, 16 - 21
 complement (集合) 补, 9

- countable 可数 (集合), 7, 17-19
- countably infinite 可数无穷 (集合), 17-19
- denumerable 可数 (集合), 17-19
- difference 差分 (集合), 9
- disjoint 不相交 (集合), 9
- empty 空 (集合), 8
- equality 对等 (集合), 8, 11
- infinite 无穷 (集合), 17
- intersection 交叉 (集合), 9
- lookahead 预读 (集合), 572, 575, 589
- partition 分割 (集合), 9
- power 幂 (集合), 9, 20
- proper subset of 真子 (集), 9
- regular 正则 (集合), 49-50
- subset of 子 (集), 8
- uncountable 不可计算 (集合), 7, 17
- union 并 (集), 9
- Shift 移进, 556
- Simple cycle 简单循环, 33
- Simple induction 简单归纳 30
- Space bounded Turing machine 空间受限图灵机, 534
- Space complexity 空间复杂性, 532-535
- Speedup Theorem 加速定理, 448-451
- Stack 栈
 - acceptance by empty stack 被空栈接收, 230
 - alphabet 栈字母表, 222
- Standard Turing machine 标准图灵机, 255-257
- State 状态, 145-147
 - accepting 接收 (状态), 147
 - equivalent 等价 (状态), 178
 - start 开始 (状态), 147, 256
- State diagram 状态图, 146-147
 - of deterministic finite automaton, 确定的有限自动机的 (状态图), 151-153
 - of multitape machine 多带机的 (状态图), 268
 - of nondeterministic finite automaton 非确定型的有限自动机的 (状态图), 163
 - of pushdown automaton 下推自动机的 (状态图), 222-223
 - of Turing machine 图灵机的 (状态图), 254
- Strictly binary tree 严格二叉树, 35-36
- String 串, 41-45
 - accepted string 已接收的串, 148, 224
 - concatenation 连接, 43-44
 - derivable 可推导的 (串), 69
 - homomorphism 同态, 219, 257
 - languages and 语言和 (串), 41, 43
 - length (串的) 长度, 43
 - lookahead 预读 (串), 572
 - null 空 (串), 42
 - prefix of (串的) 前缀, 45
 - reversal (串的) 反转, 45
 - substring 子串, 44-45
 - suffix of (串的) 后缀, 45
- Strong induction 强归纳, 30
- Strong LL(k) 强 LL(k)
 - grammar (强 LL(k)) 文法, 579-580
 - parser (强 LL(k)) 分析器, 587-588
- Subset 子集, 8
- Subset-Sum Problem 子集和问题, 473, 509-513
- Substring 子串, 44-45
- Successful computation 成功计算, 224
- Suffix 后缀, 45
- Symbol 符号
 - nonterminal 非终结 (符), 65, 69
 - terminal 终结 (符), 65
 - useful 有用 (符), 116
 - useless 无用 (符), 116
- Tape 带, 147-148, 255-256
 - multitrack 多道 (带), 263-265
 - output 输出 (带), 282-283
 - two-way infinite 双向无穷 (带), 265-268
- Tape alphabet 带字母表, 256
- Tape number 带数字, 416
- Terminal prefix 终结符前缀, 129, 557
- Terminal symbol 终结符, 65
- Termination, abnormal 终结, 257
- 3-conjunctive normal form 3-合取范式, 498
- 3-Satisfiability Problem 3-可满足性问题, 498-500
 - reductions from 从 (3-可满足性问题) 归约, 500-513
- Time complexity 时间复杂性, 442-446
 - nondeterministic 非确定型的 (时间复杂性), 466
 - properties of (时间复杂性的) 特性, 451-458
 - and representation (时间复杂性) 和表示法, 469-471
- Token 符号, 553, 567
- Top-down parser 自顶向下分析器, 555
 - breadth-first 宽度优先 (自顶向下分析器), 557-561
 - LL(k) (自顶向下分析器), 591
 - strong LL(k) 强 LL(k) (自顶向下分析器), 587-588
- Total function 全函数, 13
- Tour 周游, 359, 474
- Transition function 转换函数
 - of deterministic finite automaton 确定性有限自动机的 (转换函数), 147
 - extended 扩展的 (转换函数), 151
 - input (转换函数的) 输入, 170
 - multitape (转换函数的) 多带, 268

- multitrack (转换函数的) 多道, 264
- of NFA- λ NFA- λ 的 (转换函数), 166
- of nondeterministic finite automaton 非确定型有限自动机的 (转换函数), 163
- of nondeterministic Turing machine 非确定型图灵机的 (转换函数), 274-275
- of pushdown automaton 下推自动机的 (转换函数), 222-223
- of Turing machine 图灵机的 (转换函数), 256
- Transition table 状态转换表, 150
- Traveling Salesman Problem 巡回售货员问题, 517-518
- approximation algorithm for 用于 (巡回售货员问题) 的逼近问题, 521-523
- Tree 树, 33
- binary 二叉 (树), 35
- complete binary 完全二叉 (树), 40
- derivation 推导 (树), 71-74
- frontier of (树) 的边界, 35
- ordered 有序 (树), 33-34
- search 搜索 (树), 558-561
- strictly binary 严格二叉 (树), 35-36
- Truth assignment 真值赋值, 481-482
- Turing computable 图灵机可计算的
- function (图灵机可计算) 函数, 296
- relation (图灵机可计算) 关系, 299
- Turing machine 图灵机, 2, 255-257
- abnormal termination of (图灵机) 的异常终止, 296
- acceptance by entering 被进入方式接收, 263, 289
- acceptance by final state 被终结状态方式接收, 260
- acceptance by halting 被停机方式接收, 262-263
- arithmetization of (图灵机) 的算术化, 416-417
- atomic 原子 (图灵机), 290
- context-sensitive 上下文有关 (的图灵机), 290
- halting (图灵机) 的停机, 257
- Halting problem for 用于 (图灵机) 的停机问题, 357, 362-365
- as language acceptor 作为语言的接收器, 259-262
- as language enumerator 作为语言枚举器, 282-288
- linear speedup 线性加速, 448-451
- multitape machine 多带机, 268-274
- multitrack machine 多道机, 263-265
- nondeterministic Turing machine 非确定型图灵机, 274-282
- sequential operation of (图灵机) 的顺序操作, 301-302
- space complexity 空间复杂性, 532-535
- standard 标准 (图灵机), 255-257
- state diagram 状态图, 257
- time complexity of (图灵机) 的时间复杂性, 442-443, 466
- two-way 双向 (图灵机), 265-268
- universal 通用 (图灵机), 354-358
- Two-way Turing machine 双向图灵机, 265-268
- U
- Unary representation 一元表示, 299
- Uncomputable function 不可计算的函数, 312-313
- Uncountable set 不可数集, 7, 17
- examples of (不可数集) 的例子, 19-20
- Undecidable problem 不可判定的问题, 361
- Blank Tape Problem 空带问题, 366-368
- for context-free grammars 关于上下文无关文法的, 382-386
- Halting Problem 停机问题, 362-365
- Post Correspondence Problem 波斯特对应问题, 377-382
- Word Problem 词语问题, 373-376
- Union of sets 集合的并, 9
- Universal Turing machine 通用图灵机, 354-358
- Unrestricted grammar 无限制文法, 254, 325-332
- Useful symbol 有用符号, 116
- Useless symbol 无用符号, 116
- removal of (无用符号) 的删除, 116-121
- V
- Variable 变量
- Boolean 布尔 (变量), 481
- of a grammar 文法 (变量), 65, 68-69
- nullable 可空 (变量), 107
- reachable 可达 (变量), 119
- recursive 递归 (变量), 71, 390
- Vertex 顶点, 32
- Vertex Cover Problem 顶点覆盖问题, 500-503, 527
- Viable prefix 活前缀, 599
- W
- Well-formed formula 合式的公式, 481
- Word Problem 词语问题, 373-376